[illegible]

(12) INTERNATIONAL APPLICATION PUBLISHED UNDER THE PATENT COOPERATION TREATY (PCT)

(19) World Intellectual Property Organization
International Bureau

(43) International Publication Date
29 November 2001 (29.11.2001)

PCT

(10) International Publication Number
WO 01/90961 A2

(51) International Patent Classification⁷: G06F 17/50

(21) International Application Number: PCT/US01/09106

(22) International Filing Date: 22 March 2001 (22.03.2001)

(25) Filing Language: English

(26) Publication Language: English

(30) Priority Data:
09/577,509 24 May 2000 (24.05.2000) US

(71) Applicant: **KONINKLIJKE PHILIPS ELECTRONICS NV** [NL/NL]; Groenewoudseweg 1, NL-5621 BA Eindhoven (NL).

(71) Applicant *(for all designated States except CN, JP, KR)*: **PHILIPS ELECTRONICS NORTH AMERICA CORPORATION** [US/US]; 1000 West Maude Avenue, Sunnyvale, CA 94086-2810 (US).

(72) Inventors: **DUBOC, Jean, Francois**; 2893, Moyenne Corniche des Pugets, F-06700 Saint-Laurent-du-Var (FR). **MALLET, Frederic**; 30, Boulevard Las Palmas, F-06100 Nice (FR). **BOERI, Fernand**; 131, Promenade des Anglais, F-06200 Nice (FR).

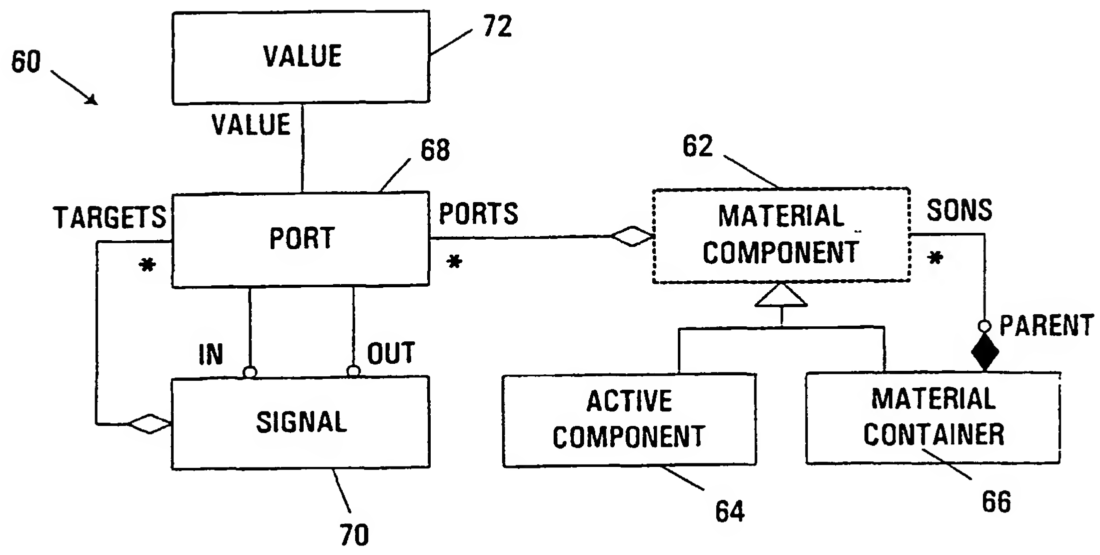
(74) Agents: **STINEBRUNER, Scott, A.** et al.; Wood, Herron & Evans, LLP, 2700 Carew Tower, Cincinnati, OH 45202 (US).

(81) Designated States *(national)*: CN, JP, KR.

(84) Designated States *(regional)*: European patent (AT, BE, CH, CY, DE, DK, ES, FI, FR, GB, GR, IE, IT, LU, MC, NL, PT, SE, TR).

**Published:**
— *without international search report and to be republished upon receipt of that report*

(54) Title: ENHANCEMENTS TO OBJECT-ORIENTED ELECTRONIC CIRCUIT DESIGN MODELING AND SIMULATION ENVIRONMENT



(57) Abstract: An apparatus, program product and method enhance the modeling and stimulation of electronic circuit designs on computers, particularly in the area of object-oriented modeling and simulation. Multiplexed communication ports (130) may be supported that support the propagation of values for multiple signals in response to single events directed to such ports (130), thus logically combining multiple signal paths together into a single logical path. Dynamic instruction decoder generation (160) may also be supported, such that an instruction definition that defines an instruction capable of being executed by a processing system model may be utilized in the configuration of an instruction decoder circuit component (150') for the processing system model to stimulate execution of the instruction.

WO 01/90961 A2

# ENHANCEMENTS TO OBJECT-ORIENTED ELECTRONIC CIRCUIT DESIGN MODELING AND SIMULATION ENVIRONMENT

## Field of the Invention

The invention is generally related to the design and modeling of integrated and other electronic circuits. More specifically, the invention is generally related to the software-based design and modeling of integrated and other electronic circuits for prototyping and performance evaluation.

## Background of the Invention

As semiconductor fabrication technology advances, designers of integrated circuits and electronic circuits incorporating the same are able to integrate more and more functions into a single integrated circuit device, or chip. As such, electronic designs that once required several integrated circuits electrically coupled to one another on a circuit board or module may now be integrated into a single integrated circuit, thereby increasing performance and reducing cost.

With increases in circuit complexity, however, the processes of designing and testing circuit designs becomes increasingly complex and time consuming. As a result, computers have become increasingly important in automating the design and testing of circuit designs.

Generally, an electronic circuit design is first defined at a relatively high level of abstraction, often without the assistance of any software design tools. Basic operational functionality and desired performance is decided upon, often by a team of designers. Once the high level specification is defined, a hardware definition of the design is developed using a hardware description language such as VHDL or Verilog. From the hardware definition, logic simulation is often performed to validate the functionality of the design. Once the functionality has been validated, physical layout of the individual circuit components in the design is performed, often with the assistance of a synthesis tool. More detailed simulation and timing analysis are then performed on the synthesized design to locate and correct any specific problems with the design. Once the design has been verified, the design is typically ready for manufacture and final testing of manufactured parts.

Substantial advances in design reuse and abstraction have permitted developers to leverage pre-existing designs in the development of new designs. Pre-designed and pre-tested circuit components, for example, may be modified and/or assembled together to define an integrated circuit design, thus eliminating the requirement for a developer to start each design from scratch. Also, with reusable circuit components having already been tested and often optimized for particular functions, many of the difficulties associated with synthesis and simulation may also be avoided.

The functionality encapsulated within reusable circuit components has also increased substantially. Whereas reusable circuit components were once limited to relatively simple circuit blocks that represented simple logical components such as multiplexers and registers, more advanced components have been developed incorporating relatively complex circuit components such as entire processor cores, memories, digital signal processor cores, etc.

Abstraction and design reuse are both important contributors to maintaining reasonable development cycles in the face of ever-increasingly complex circuit designs. Nonetheless, with the rapid pace of technological change, the time-to-market for integrated circuit designs continues to be driven downward, thus necessitating further advancements in design automation.

Development cycles are significantly affected by changes to functionality and implementation, as changes often necessitate additional efforts in reworking a design and retesting the modified design. As a result, it is often extremely desirable to evaluate and settle upon basic design parameters as early as possible in the development process.

To support the early evaluation of design parameters, object-oriented rapid prototyping has been proposed to provide early-process evaluation of integrated circuit designs, particularly of digital architectures incorporating complex features such as embedded processor cores, digital signal processor (DSP) cores, memory systems, etc. With object-oriented rapid prototyping, circuit components are modeled using an object-oriented software language. Traditional object-oriented concepts such as abstraction, encapsulation, reuse, inheritance, and polymorphism are used to build a library of circuit components that are ultimately based upon one or more generic component objects. To create a design, therefore, existing components are assembled together and/or new components are constructed by modifying or extending the functionality of existing components.

Object-oriented rapid prototyping and other modeling and simulation environments will likely provide circuit developers with the ability to quickly evaluate and define the high level specifications of new circuit designs extremely early in the design process, thus serving to shorten development cycles in many instances. Proposed software tools to implement object-oriented modeling and simulation, however, are limited in functionality. As such, a significant need exists in the art for a software automation tools that implement and enhance object-oriented modeling and simulation concepts to further accelerate the development of electronic circuit designs and the like.

## Summary of the Invention

The invention addresses these and other problems associated with the prior art by providing an apparatus, program product and method that enhance the modeling and simulation of electronic circuit designs on computers, particularly in the area of object-oriented modeling and simulation.

Consistent with one aspect of the invention, modeling and simulation are enhanced through the support for multiplexed ports within circuit components in an electronic design model. A multiplexed port consistent with the invention supports the propagation of values for multiple signals in response to an event directed to the port, thus logically combining multiple signal paths together into a single logical path. Among other benefits, when a model is represented in a graphical user interface environment of a computer, the single logical path representing the multiplexed port may be displayed in lieu of separate paths for individual signals capable of being propagated through the port, thereby reducing display clutter and facilitating comprehension of the model.

As an example, in a method consistent with this aspect of the invention, an electronic circuit design is modeled on a computer by constructing the model from first and second object-oriented circuit components that respectively include first and second ports, and interfacing the first and second circuit components to one another through the first and second ports. The first and second ports are configured to propagate both a first value for a first signal and a second value for a second signal between the first and second ports responsive to an event directed to at least one of the first and second ports.

Consistent with another aspect of the invention, modeling and simulation of an electronic circuit design are enhanced by providing a dynamically generated instruction decoder circuit component for use in decoding instructions to be executed within a processing system model representative of the electronic circuit design. Generation of an instruction decoder circuit component consistent with the invention includes receiving at least one instruction definition that defines an instruction capable of being executed by a processing system model, and processing the instruction definition to configure an instruction decoder circuit component for the processing system model to simulate execution of the instruction.

These and other advantages and features, which characterize the invention, are set forth in the claims annexed hereto and forming a further part hereof. However, for a better understanding of the invention, and of the advantages and objectives attained through its use, reference should be made to the Drawings, and to the accompanying descriptive matter, in which there is described exemplary embodiments of the invention.

## Brief Description of the Drawings

FIGURE 1 is a block diagram of a computer system for use in performing object-oriented modeling and simulation consistent with the invention.

FIGURE 2 is a block diagram of an exemplary hardware and software environment for a computer from the computer system of Fig. 1.

FIGURE 3 is a block diagram illustrating the primary components in a rapid prototyping environment implemented in the computer of Fig. 2.

FIGURE 4 is an object diagram of a generic component model for use in the rapid prototyping environment of Fig. 3.

FIGURE 5 is a block diagram of a behavioral representation of an exemplary load/store register circuit component based on the generic component model of Fig. 4.

FIGURE 6 is a block diagram of a structural representation of the exemplary load/store register circuit component of Fig. 5.

FIGURE 7 is a flowchart illustrating the sequence of operations utilized in prototyping an architecture using the rapid prototyping environment of Fig. 3.

FIGURE 8 is a block diagram of an exemplary processing system circuit component capable of being modeled by the rapid prototyping environment of Fig. 3.

FIGURE 9 is a block diagram of the processor circuit component from the exemplary processing system circuit component of Fig. 8.

FIGURE 10 is a block diagram of an exemplary simulation window displayed by the rapid prototyping environment of Fig. 3 during development of the exemplary processing system circuit component of Fig. 8.

FIGURE 11 is a flowchart illustrating the sequence of operations utilized in configuring a port using the rapid prototyping environment of Fig. 3.

FIGURE 12 is a block diagram of an icon representation of the instruction decoder from the processor circuit component of Fig. 9, including a generic decoder circuit component embedded therein.

FIGURE 13 is a flowchart illustrating the sequence of operations utilized in customizing an instruction decoder using the rapid prototyping environment of Fig. 3.

FIGURE 14 is a flowchart illustrating the program flow of the generate decoder routine referenced in Fig. 13.

FIGURE 15 is a block diagram of the icon representation of the generic decoder circuit component of Fig. 12, after dynamic generation to incorporate an exemplary instruction set using the generate decoder routine of Fig. 14.

FIGURE 16 is a block diagram of exemplary data and program memory windows displayed by the rapid prototyping environment of Fig. 3 during simulation.

FIGURE 17 is a block diagram of an exemplary virtual component capable of being modeled by the rapid prototyping environment of Fig. 3.

## Detailed Description

### Hardware and Software Environment

Turning to the Drawings, wherein like numbers denote like parts throughout the several views, Fig. 1 illustrates a computer system 10 for use in performing object-oriented modeling and simulation consistent with the invention. Computer system 10 is illustrated as a networked computer system including one or more client computers 12, 14 and 20 (e.g., desktop or PC-based computers, workstations, etc.) coupled to server 16 (e.g., a PC-based server, a minicomputer, a midrange computer, a mainframe computer, etc.) through a network 18. Network 18 may represent practically any type of networked interconnection, including but not limited to local-area, wide-area, wireless, and public networks (e.g., the Internet). Moreover, any number of computers and other devices may be networked through network 18, e.g., multiple servers.

Client computer 20, which may be similar to computers 12, 14, may include a central processing unit (CPU) 21; a number of peripheral components such as a computer display 22; a storage device 23; a printer 24; and various input devices (e.g., a mouse 26 and keyboard 27), among others. Server computer 16 may be similarly configured, albeit typically with greater processing performance and storage capacity, as is well known in the art.

Fig. 2 illustrates in another way an exemplary hardware and software environment for an apparatus 30 consistent with the invention. For the purposes of the invention, apparatus 30 may represent practically any type of computer, computer system or other programmable electronic device, including a client computer (e.g., similar to computers 12, 14 and 20 of Fig. 1), a server computer (e.g., similar to server 16 of Fig. 1), a portable computer, a handheld computer, an embedded controller, etc. Apparatus 30 may be coupled in a network as shown in Fig. 1, or may be a stand-alone device in the alternative. Apparatus 30 will hereinafter also be referred to as a "computer", although it should be appreciated the term "apparatus" may also include other suitable programmable electronic devices consistent with the invention.

Computer 30 typically includes at least one processor 31 coupled to a memory 32. Processor 31 may represent one or more processors (e.g., microprocessors), and memory 32 may represent the random access memory (RAM) devices comprising the main storage of computer 30, as well as any supplemental levels of memory, e.g., cache memories, non-volatile or backup memories (e.g., programmable or flash memories), read-only memories, etc. In addition, memory 32 may be considered to include memory storage physically located elsewhere in computer 30, e.g., any cache memory in a processor 31, as well as any storage capacity used as a virtual memory, e.g., as stored on a mass storage device 35 or on another computer coupled to computer 30 via network 36.

Computer 30 also typically receives a number of inputs and outputs for communicating information externally. For interface with a user or operator, computer 30 typically includes one or more user input devices 33 (e.g., a keyboard, a mouse, a trackball, a joystick, a touchpad, and/or a microphone, among others) and a display 34 (e.g., a CRT monitor, an LCD display panel, and/or a speaker, among others).

For additional storage, computer 30 may also include one or more mass storage devices 35, e.g., a floppy or other removable disk drive, a hard disk drive, a direct access storage device (DASD), an optical drive (e.g., a CD drive, a DVD drive, etc.), and/or a tape drive, among others. Furthermore, computer 30 may include an interface with one or more networks 36 (e.g., a LAN, a WAN, a wireless network, and/or the Internet, among others) to permit the communication of information with other computers coupled to the network. It should be appreciated that computer 30 typically includes suitable analog and/or digital interfaces between processor 31 and each of components 32, 33, 34, 35 and 36 as is well known in the art.

Computer 30 operates under the control of an operating system 40, and executes or otherwise relies upon various computer software applications, components, programs, objects, modules, data structures, etc. (e.g., a rapid prototyping modeling an simulation environment 41 incorporating a simulator 42, compiler 44, user interface 46 and component libraries 48, among others). Moreover, various applications, components, programs, objects, modules, etc. may also execute on one or more processors in another computer coupled to computer 30 via a network 36, e.g., in a distributed or client-server computing environment, whereby the processing required to implement the functions of a computer program may be allocated to multiple computers over a network.

In general, the routines executed to implement the embodiments of the invention, whether implemented as part of an operating system or a specific application, component, program, object, module or sequence of instructions will be referred to herein as "computer programs", or simply "programs". The computer programs typically comprise one or more instructions that are resident at various times in various memory and storage devices in a computer, and that, when read and executed by one or more processors in a computer, cause that computer to perform the steps necessary to execute steps or elements embodying the various aspects of the invention. Moreover, while the invention has and hereinafter will be described in the context of fully functioning computers and computer systems, those skilled in the art will appreciate that the various embodiments of the invention are capable of being distributed as a program product in a variety of forms, and that the invention applies equally regardless of the particular type of signal bearing media used to actually carry out the distribution. Examples of signal bearing media include but are not limited to recordable type media such as volatile and non-volatile memory devices, floppy and other removable disks, hard disk drives, magnetic tape, optical disks (e.g., CD-ROM's, DVD's, etc.), among others, and transmission type media such as digital and analog communication links.

In addition, various programs described hereinafter may be identified based upon the application for which they are implemented in a specific embodiment of the invention. However, it should be appreciated that any particular program nomenclature that follows is used merely for convenience, and thus the invention should not be limited to use solely in any specific application identified and/or implied by such nomenclature.

Those skilled in the art will recognize that the exemplary environments illustrated in Figs. 1 and 2 are not intended to limit the present invention. Indeed, those skilled in the art will recognize that other alternative hardware and/or software environments may be used without departing from the scope of the invention.

## Enhanced Object-Oriented Rapid Prototyping Environment

Fig. 3 illustrates the primary components for implementing a rapid prototyping environment 41 incorporating object-oriented modeling and simulation consistent with the invention. In the illustrated implementation, rapid prototyping environment 41 is implemented using an object-oriented programming language such as the Java programming language from Sun Microsystems. It will be appreciated that the rapid prototyping environment may be implemented in other object-oriented programming languages, e.g., c++, Smalltalk, etc. It will also be appreciated that the invention may have applicability in modeling and/or simulation applications other than in rapid prototyping. Therefore, the invention is not limited to the particular implementations discussed herein.

As shown in Fig. 3, in rapid prototyping environment 41, a Java-based simulator 42 interacts with a Java compiler 44, a user interface 46 and component libraries 48. In addition, within the user interface is an architecture window component 50 and a debug window component 52.

Primary management of modeling and simulation functionality in environment 41 is handled by Java simulator 42. To initiate modeling of a new or existing integrated circuit design, or architecture, Java simulator 42 invokes an architecture window component 50 to provide a user interface for interacting with a developer to construct a model representative of the desired architecture to be simulated.

The architecture window is capable of retrieving circuit components from component libraries 48, assembling such components together and/or modifying instances of such components, and generating new components that are stored back into the component libraries. Based upon the object-oriented framework, each circuit component in the library is typically represented by a collection of child components. The links between these components are provided by the libraries to the Java compiler 44, which, in response to a compile command generated by a user via the architecture window 50, results in the generation of a compiled model by Java compiler 44. It will therefore be appreciated that a circuit component in this context may be represented by any data structure, source code, interpreted code and/or executable code assists in functionally modeling the operation of an electronic circuit or portion thereof.

Once the model to be simulated has been compiled by Java compiler 44, a simulation is capable of being run on the model by Java simulator 42. In addition to the compiled model, Java simulator 42 may also receive instruction decoder information suitable for simulating a desired instruction set, as well as simulation parameters indicating what information should be observed simulation. Based upon such simulation parameters, debug information is generated during simulation, and output to debug window component 52 for display to a user.

In the illustrated implementation, both the environmental and development components (e.g.,, simulator 42 and windows 50, 52), and the circuit components representative of the model to be simulated (e.g., within component libraries 48), are implemented using the Java programming language, and are compiled and interpreted on a conventional Java-based platform. It will be appreciated, however, that other programming environments may be used to implement the underlying modeling and/or simulation environment, including non-object-oriented platforms.

## Object-Oriented Modeling of Circuit Components

As mentioned above, models developed in environment 41 are principally constructed by assembling together a multitude of circuit components representing the underlying functionality of the models. In the illustrated embodiment, all components used in a model are based upon a generic component model, e.g., generic component model 60 represented in a UML presentation in Fig. 4.

Model 60 describes the minimum interface between components and constitutes a recursive model container-component between a material component 62, active component 64 and material container 66. The material container only serves to encapsulate material components and signals, allowing a structural description for material components. Active components represent components incorporating behavior aspects of a component, herein referred to as services.

Material components are interfaced with one another via one or more service ports 68, which are used to pass signals 70 to other components, with values 72 specifying the actual data to be communicated between connected ports. Values can be constrained to specific data types, e.g., instructions, integers, events, bit values, hardware levels, etc.; however, at the generic model level, the type of data specified by a value is not type-specific.

Based upon this generic model, circuit components may be developed by extending the model and defining services associated with ports for modeling the behavioral performance of such components. Furthermore, circuit components may be defined structurally to incorporate child components that define functions, or services, capable of being accessed by such parent components. Circuit components may therefore be represented both structurally and behaviorally depending upon the aggregated services and child components encapsulated thereby.

For example, Fig. 5 illustrates the interface of a load/store register 80 incorporating a plurality of ports 82 interfaced with a behavior, represented at 84. The behavior may be described either in a structural manner or in a functional manner.

For example, as shown in Fig. 6, a structural representation 90 of the same load/store register is illustrated, in which the ports 92 corresponding to ports 82 of Fig. 5 are shown interfaced with a pair of registers 94 representing the underlying behavior of the load/store register. Each register component 94 represents a more elementary component utilized by the load/store register to implement load/store functionality. Specifically, gate ports of the register components 94 are coupled to load and store ports 92 in register 90, such that load and store signals passed to the load/store component are routed to the appropriate register 94 to implement the desired operational behavior for a load/store register.

An alternate manner of representing the behavior of the exemplary load/store register is via a functional manner, e.g., via the Java programming language. For example, Table I below illustrates representative Java source code for the load/store register of Fig. 5:

**Table I: Example Load/Store Register B havioral Description**

```
package component.basic;
public LSRegister extends model.simulation.EdgeComponent {
        public LSRegister() {
                addInactivePort("in",LEFT);
                addInactivePort("out",LEFT);
                addRisingEdgePort("load",TOP).setService("load");
                addRisingEdgePort("store",TOP).setService("store");
        }
        public void load() {
                setPriority(1);
                emit("out",read("value"));
        }
        public void store() {
                setPriority(0);
                emit("value",read("in"));
        }
}
```

In the aforementioned description, the EdgeComponent class inherits from the material component class, and describes the behavior that occurs when an edge event occurs on one of the ports of a component. In the EdgeComponent class, each port may be assigned a sensitivity to different types of events, e.g., insensitive, sensitive on a rising edge, sensitive on a falling edge, etc. Various *addPort()* methods are supported to register different types of ports in the component, as well as to associate services with such ports (via a *setService()* method). Supplied to each *addPort()* method is a port name along with a directional indicators (e.g., LEFT, RIGHT, TOP, BOTTOM), which is used to indicate where on an icon representation of a component the port should be displayed.

When an event occurs on a sensitive port of an EdgeComponent component, the service (i.e., the method specified in the *setService()* call) is executed. Sensitivity therefore describes the sequence of events needed to make a component reacting -- i.e., the execution of an attached service. The composition of several services constitutes the behavior of a component.

As an example of appropriate services to be utilized in the implementation of a component, the load and store services shown in Table I above each utilize a *priority()* method and an *emit()* method. The *priority()* method defines relative priorities for different services, for the purpose of conflict resolution. The *emit()* method is provided with the name of a port to drive, along with a signal value to be driven on that port (here, the current state of another port in the component, accessible via the *read()* method).

Typically, there is no type associated to ports, and as such, there is no constraint on links between ports, so that each port and each component may receive any data type. Each data inherits from the abstract class value and has the ability to mute to match another attempt type, thereby increasing reusability of components.

As an example, the load/store register has some constraints. The "IN" port needs a producer to serve some values, so the "IN" port cannot be connected to a data bus where no potential producer may emit some data. In the same way, a system may deduce that load and store ports have to be connected to ports able to provide some level value since they are edge sensitive ports. This is the first step of structural verification.

Further, since inheritance and subtyping may be used to describe values, more than one operator may apply for a specific operation. As an example, an arithmetical and logical unit (ALU) may be defined as a component owning two input data ports, one input control port and two output ports, which represent the result and flag result. Then, whatever effective driven data types are, the component has to compute the required operation. By example, an addition can be seen as integer addition, float addition, string concatenation, or others. Data should have the ability to mute in accordance with a defined sub-type relation. In addition, the right operator should be capable of being inferred dynamically in simulation. This may be performed by parsing the sub-type tree selecting the lowest operator which matches parameter requirements. If no operator can be inferred, then the "NONE" value may be computed and a message sent to the developer. This may be done due to the lack of an adapted description for the concerned operator requiring completion of the description. This could also indicate that some data paths are wrong and should be corrected.

Since a component is a set of ports and a set of services, inheritance of hardware components typically means inheritance of ports and inheritance of the services that can be overridden. Furthermore, as will be discussed in greater detail below, ports may be configured to provide sequential and parallel composition of multiple services provided by contained components, such that multiple signals are capable of being propagated through a port in a containing component in response to a given event, and subsequently distributed to the appropriate encapsulated components for performing multiple services in response to the event. This composition may be abstracted by a textual identifier which can then be considered as a higher level service.

With the aforementioned object framework, it is possible to inherit behavior from a structural component, providing a more powerful extension than simple structural inheritance. Inheritance used to extend components may be explained, for example, by definition of a shift register based upon a basic register component. A basic register may be defined, for example, as shown in Table II below, including a "LOAD" service to load an input value onto an output port, and a "RESET" service to set the output to zero:

**Table II: Example Basic Register Behavioral Description**

```
package component.basic;
public Register extends model.simulation.EdgeComponent {
    public Register() {
        addInactivePort("in",LEFT);
        addInactivePort("out",RIGHT);
        addRisingEdgePort("rst",TOP).setService("reset");
        addRisingEdgePort("ld",BOTTOM).setService("load");
    }
    public void reset() {
        setPriority(1);
        emit("out",0);
    }
    public void load() {
        setPriority(0);
        emit("out",read("in"));
} }
```

To define a shift register inheriting from the basic register, the shift register class is provided with an additional "SHIFT" service, which shifts a memorized value one bit to the left. Additional "CIN" and "COUT" ports are provided to handle carry bits. After shifting, a new high significant bit for the memorized value is defined by the value assigned to a "CIN" port, with the old low significant bit assigned to a "COUT" port. One suitable behavioral description of a shift register which extends the register description of Table II is shown below in Table III:

**Table III: Example Inherited Shift Register Behavioral Description**

```
package component.basic;
public ShiftRegister extends Register {
    private int size;
    public ShiftRegister(int size) {
        super();         // "in", "out", "rst", "ld" ports are inherited
        this.size=size;
        addRisingEdgePort("shr",BOTTOM).setService("shift");
        addInactivePort("cin",LEFT);
        addInactivePort("cout",RIGHT);
    }
    public ShiftRegister(8) {this(8);} // default size is 8
    public void shift() {
        setPriority(-1);         // lower priority than load() or reset()
        IntValue content = read("out");
        emit("cout",content.trunc(1).getLevelValue());
        content = content.shr().add(read("cin").getIntValue().shl(size-1));
        emit("out",content);
    }
}
```

The constructor for the shift register class has to define the new ports and the new shift service. The other services of the basic register are inherited. In addition, it will be appreciated that the services of the basic register component could also be overridden if necessary to implement the functionality of another component inheriting from the register component.

Additional discussion of the object-oriented modeling and simulation as described herein is provided in Mallet F.; Boéri F.; and Duboc J-F, 1998, "Hardware Modeling and Simulation Using an Object-Oriented Method", *Proceedings of the European Simulation Multiconference*, June 1998, pp. 166-168, the disclosure of which is incorporated by reference herein.

## Digital Architecture Prototyping with Object-Oriented Rapid Prototyping Environment

A sequence of operations for prototyping a programmable digital architecture using rapid prototyping environment 41 described herein is illustrated at 95 in Fig. 7. Generally, a new model representative of a desired design or architecture is initially constructed from existing library components as illustrated in block 96. Given that the desired architecture is programmable, typically a processor core or like component suitable for executing a program of instructions is incorporated into the model.

Subsequent to the design of the model, any program and/or data memories are initialized for use in the simulation, as represented in block 97. In particular, for the programmable digital architecture, a program of instructions typically must be loaded into a memory to be executed during simulation. As will be discussed in greater detail below, memory contents may be defined via textual files, tab-delimited files, or in other manners of representing arrayed data in a file or database.

Moreover, as will become more apparent below, for program instructions, it may not be necessary from a functional standpoint to store the actual binary opcodes and associated binary data that would be stored in an actual memory and executed by an actual hardware implementation of a modeled digital architecture. Instead, for high level modeling and simulation applications it may be sufficient to simply store text assembly-language representations of the instructions in a memory, and then interpret those assembly-language representations during simulation to accurately simulate the execution of such instructions.

Once any memories have been initialized with sample data, a user specifies the registers and/or ports to monitor during simulation, as represented in block 98. As will also become more apparent below, dedicated display port components that function only as output devices for displaying data during simulation may be incorporated into a model to assist in debugging the model. In addition, it may be desirable to support direct output of state information from selected components in a model by providing an output service in each component that is manually selectable by a user to determine whether the current value of a port or specific variable in a component should be output for display during simulation. Other manners of outputting simulation information may be used in the alternative.

Once the information to be displayed has been specified, a simulation is run in block 99. Typically, such simulation proceeds in much the same manner as other forms of event-based logic simulators. Simulation cycles are based upon clock cycles, with events propagated throughout a model during each clock cycle to simulate the logical function of the model. User settings may be configured to control the number of cycles and delays between pauses in the simulation, with additional user input capable of quickly stepping through a simulation, backing up in the simulation, etc.

Next, as shown in block 100, upon completion of the simulation, it is determined whether an acceptable simulation result has been obtained--that is, whether the model meets the desired functional performance of the architecture.

If not, the model may need to be modified through the addition of new components, and/or the modification of existing components, as shown at block 101. The simulation may then be rerun until an acceptable result is obtained. Upon receiving an acceptable result, the created model is typically archived as a new component in the library, as shown at block 102, and prototyping of the architecture is then complete.

To further explain the modeling and simulation process described herein, Fig. 8 illustrates an exemplary processing system architecture 104 capable of being modeled and simulated by rapid prototyping environment 41 described herein. The processing system is comprised of a CPU or processor component 105 and a memory 106 that contains the program and data acted upon by the CPU. Both components 105 and 106 are capable of receiving a RESET signal, and the CPU is configured to receive a CLOCK signal for use in simulating the processing system environment. The CPU is capable of sending an address over an address (ADR) bus, accompanied by a READ or WRITE signal, with data transferred between the memory and the CPU over a data bus. The memory includes two outputs UNI and UNI_I, which are optionally provided to indicate, respectively, whether the memory is initialized and whether a program has been loaded into the memory. Such outputs may be used to indicated to a developer that the memory at least has to be loaded with a program to perform simulation.

Fig. 9 illustrates in greater detail the architecture of the CPU 105. The computation part of the CPU is formed by an ALU 107 and an accumulator register 108. The control part of the CPU includes a program counter (PC) 110 used to generate an instruction address representative of the address of the current instruction being executed by the processor. When an instruction is fetched, it is provided to an instruction register (IR) 112 from DATA IN lines of the data bus, where instruction decoding is performed. After the instruction is decoded, the instruction register sends command signals to the various blocks of the architecture, including ALU 107, PC 110 and a controller 114.

The data bus is coupled to the CPU with a DATA IN coupled to the instruction register 112, as well as to an input of the ALU 107. Also provided to the ALU is a control signal CODEOP from the instruction register, as well as an output of the contents of the accumulator register (ACCU) 108. The result of the ALU operation is fed back via an output to the accumulator register 108, as well as to a buffer 118 that couples the output of the ALU to DATA OUT lines of the data bus. The ALU also provides a zero flag signal to controller 114.

Controller 114 outputs a number of control signals to control the other blocks within the CPU. The primary task of the controller is to determine if a read or write is to be performed with the memory in a given clock cycle and if the read or write will be a data exchange (where the address emanates from instruction register 112), or an instruction read (where the address emanates from the program counter 110). Selection of the appropriate driver for the address bus is made by controller 114 via assertion of a VAL_DATA select signal to a multiplexer 116, which multiplexes the outputs of the instruction register and program counter to the address bus ADR.

In addition, a LOAD_ACCU signal controls the latching of the accumulator register 108, and an ENABLE_DATA signal enables capture of data output by ALU 107 by buffer 118. A LOAD_INST signal controls the latching of a new instruction by the instruction register 112, while INC_PC and LOAD_PC signals are provided to program counter 110 to respectively increment the PC and load the PC with a new address responsive to a decoded instruction. Controller 114 also outputs WRITE and READ signals external to the CPU responsive to the decoded instruction.

Controller 114 receives the external clock signal CLK for the CPU. In addition, each of the instruction register 112, program counter 110, accumulator 108 and controller 114 are responsive to the RESET signal provided to the CPU.

Any number of computer user interfaces may be utilized to assist a developer in developing an appropriate object-oriented model of the aforementioned processing system consistent with the invention. For example, it may be desirable to utilize a graphical user interface, e.g., using the Java abstract windowing toolkit (AWT) to support the assembly of components from a library into a desired design using a visual drag-and-drop metaphor. Using such an environment, components may be represented by icons, with drag-and-drop functionality utilized to drop components from various libraries onto a workspace. Moreover, components may be configured simply by clicking on the components and using pop-up menus, dialog boxes, and other user interface controls to configure options and other configuration parameters as desired. Interconnection of components may be performed by clicking on port controls displayed on component icons, with the interconnections represented by lines or other suitable graphical representations.

For example, Fig. 10 illustrates a representative architecture window 50 from rapid prototyping environment 41 of Fig. 3, illustrating a graphical representation of a portion of an object-oriented model of processing system 104 of Fig. 8. Illustrated within architecture window 50 is a workspace 120 and a plurality of tabs 122 representing various available component libraries that a user may select to locate appropriate components to be dropped into an architecture model. Displayed within workspace 120 are graphical icons 124, 126 and 128, respectively representing the program counter 110, instruction register 112 and controller 114 in the exemplary processing system. Icons 124-128 each include port controls 130 representing each of the ports supported by the associated components. Interconnections between ports on different components are represented graphically by interconnect elements such as lines 132 extending between port controls on different icons.

In addition to circuit components, it may also be desirable to incorporate additional controls or icons in a model to assist in simulation of the model. For example, components such as manually-actuated switches, represented at 134, may be used to manually control the values stored at different ports before or during simulation. Control 134, for example, may be utilized to generate a hard reset during simulation to return the simulated model to an initial state.

In addition, output components, e.g., display component 136, may be incorporated into a model and tied to certain ports to output the current signal applied at each port during simulation. Control 136, for example, represents the output of "ADD" value over the CODEOP control line issued by the instruction register.

It will be appreciated that implementation of the aforementioned graphical user interface environment would be well within the ability of one of ordinary skill in the art having the benefit of the instant disclosure. Additional graphical functionalities supported in Java and other application environments may also be used in some applications.

Moreover, it will be appreciated that various software development methodologies such as forms-based programming, rapid application development programming, etc. may be utilized to enhance the functionality supported by the rapid prototyping environment discussed herein. For example, selection of any icon by a user may result in the generation of a dialog box incorporating a table of configurable values and other parameters that a user may modify to customize a particular component as desired. For example, for a memory component, it may be desirable to enable a user to specify the size of a memory, the address space of the memory, the speed of the memory, the type of the memory, etc., via dialog box selections. In addition, it may be desirable to associate an external file, e.g., a text file, to provide a sample data to be used during simulation as the contents of the memory. It may also be desirable to support the unique naming of components so that a user can control the naming of instanced components as desired.

Generally, in response to the modification of a component via the modification of parameters via the graphical user interface, the Java language specification for the component is typically dynamically modified by the Java simulator. Doing so is analogous to dynamic code generation in visual programming environments, the use of configuration of which is well known in the art.

It should also be appreciated that various component library management operations may be provided consistent with the invention. As with various visual or rapid application development tools known for Java and other object-oriented languages, management of objects may be supported in a number of manners. For example, objects may be organized by bookmarks. In addition, different icons may be utilized in association with a particular component. Editing of icons may also be supported.

To enhance modeling and simulation of digital architectures, a number of unique enhancements are provided in rapid prototyping environment 41. Notable among such enhancements are port management and multiplexing, and dynamic instruction decoder generation. Each of these enhancements are discussed in greater detail below.

## Port Management

As discussed above, one specific functionality supported by the rapid prototyping environment is that of port management, for managing the transmission of data between components in a model under simulation. In the illustrated embodiment, ports are data type-independent constructs utilized to transmit values as signals between components, and in response thereto, to initiate the performance of services in the components. A basic port type is defined, from which a number of child port classes may be defined. In the illustrated implementation, for example, service ports may be defined as input ports, output ports, inout (bidirectional) ports and communication ports. As will become more apparent below, communication ports are an important aspect to providing a manageable development environment, as such ports permit the multiplexing of multiple signals across a common pathway between components.

As an example, hardware designs typically include a number of enabling or chip-select signals, configuration signals, compatibility signals, etc. that do not intervene in performance evaluation. Control signals also may be grouped with common properties, e.g., to allow selection between different paths acting on a multiplexer. When modeling in a hierarchical manner as described herein, such signals can generate a number of intermediate ports and signals. Defining a multitude of ports for carrying individual signals, and connecting all of such ports between components in the graphical environment, would result in an excessive amount of information displayed in the architecture window and clutter the display with a large number of lines that would make comprehension of the model difficult.

Therefore, to address this difficulty, communication ports are defined to incorporate the propagation of multiple signals within a given port such that a single coupling of ports in connected components can be used to represent the transmission of multiple signals between the components. An example of one situation in which the use of multiplexed communication ports is desirable is in coupling an instruction decoder to a register that incorporates multiple registers capable of being selected as sources or destinations of arithmetic operations by an ALU. By multiplexing both source selection signals and destination signals into a single communication port, individual ports for selection of each source and destination are not required, nor are individual lines extending between such ports. It will be appreciated that suitable interconnections within the register file and the decoder would still be required, e.g., for the register file, to route the register selection signals to the appropriate registers within the register file. Nonetheless, when the register file is displayed as a single icon (with the individual registers encapsulated in the file), significant display clutter is avoided, as is excessive effort in interconnecting the decoder and register file when such components are dropped into a model.

In the illustrated embodiment, each port incorporates a port name, an icon position (representing which edge of an icon the port is located), and a port type (here in, out, inout or communication). Each port is associated with one or more events (i.e., "1", "0", "rising edge", or "falling edge") generated by one or more event generators, and with a value to be propagated over the port in response to the event. The event name, for example, may be the name of a signal or clock.

Fig. 11, for example, illustrates a configure port routine 140 executed by the rapid prototyping environment in response to an attempt to configure a port for a particular component. Routine 140 may be executed, for example, in response to an attempt to create a new port by selecting a component and activating an add port operation therefor. Routine 140 may also be executed for an existing port in response to a user request to edit an existing port selected by the user.

Routine 140 begins in block 142 by obtaining the port name, icon position and type of the port (in, out, inout, communication) from the user. Next, block 144 obtains an event name, event value and propagate value for actuating the port. As discussed above, the event name refers to a signal whose status is polled to determine whether to propagate a value through the port. The event value refers to a trigger condition, in this case the value of the identified signal that will result in value propagation, e.g., when the signal is at a high or low logic level, or when a transition to a rising or falling edge is detected. The propagate value refers to the value to output through the port in response to a triggered event.

As discussed above, the propagate value may not be type-specific, and may incorporate a string representing the data to be transferred over the port. In the alternative, the value to be propagated may be a logic level (e.g., high or low voltage), or a rising or falling edge, among others.

As illustrated at block 146, another possible value for the propagate value is a "multiplex" value, representing a multiplexed communication port that a user desires to multiplex several signals over as one communication signal. In response to the detection of a multiplex request, block 146 passes control to block 148 to obtain from the user one or more name/value pairs identifying the signals and the values thereof to propagate concurrently in response to the selected event. Thus, for each signal to be propagated, the user lists the signal along with a value to be propagated for that signal. Configuration of the port is then complete. And returning to block 146, if the propagate value is not "multiplex", block 148 is bypassed, and configuration of the port is complete.

It will be appreciated that a user interface (e.g., a graphical user interface) is supported to facilitate management and configuration of ports. For example, a user may be required to select a "manage" button after selection of a port, and then utilize list boxes, check boxes or other controls within a dialog box to select various available signals, event conditions and values to propagate. Moreover, selection of the multiplex option may be via a button disposed in a dialog box selected when a user is managing a port. Other user interface controls may be used in the alternative.

During simulation, events, e.g., representing the rising or falling edges of clock signals, reset signals, or other input signals, are propagated to ports to initiate propagation of signal values through selected ports. Whether a port is a multiplexed or non-multiplexed port, detection and triggering of events is typically the same. Instead of propagating a single signal value in response to a triggering event (as with a non-multiplexed port), a multiplexed port typically propagates values for multiple signals in response to the same event.

When used in connection with the above-described generic component model, multiple control ports are capable of being abstracted in a structural component to simplify the model and reduce the number of displayed ports and signal paths. Moreover, ports are capable of transmitting signals as symbols or tokens, rather than in the form of binary logic levels, thus providing greater intuitiveness and user comprehension during simulation. As an example, for the exemplary dynamically generated instruction decoder discussed below, instructions to be decoded may be sent to the decoder as symbols or strings, e.g., "add r0, r1, r3", rather than sending actual binary opcodes. Since the underlying binary codes are not relevant to high-level performance evaluation, supporting the transmission of instructions as assembly-language strings significantly simplifies debugging during simulation.

In general, interconnection of multiplexed communication ports is represented on a display of a model via a single interconnect element, e.g., a single line (e.g., the interconnect format shown in Fig. 10. It will be appreciated that other display elements that represent interconnection between two components may be used in the alternative, and that such display elements may be capable of indicating that an interconnection is a multiplexed path rather than an interconnect for a single signal.

It will also be appreciated that typically a circuit component receiving multiplexed signals from a multiplexed port is required to implement an internal component to de-multiplex the signals and pass such signals to appropriate components within the receiving component. Combinatorial behavior, for example, may be implemented within a demultiplexing component, such that any change to a multiplexed input for the demultiplexing component is immediately transmitted to specific outputs defined for the demultiplexing component.

## Dynamic Instruction Decoder Generation

The aforementioned service port concept may also be utilized in the modeling of the instruction decoder component of a processor model. In particular, when developing a model of a processor architecture as with the exemplary architecture described above, an important component that is often implementation-specific is an instruction decoder, which is used to execute instructions from a program memory and control the computation units and the control units in a CPU to specifically configure the CPU to execute instructions received by the decoder. In the illustrated implementation, a unique dynamic decoder generation process is supported whereby a custom decoder may be dynamically generated in response to a textual definition of an instruction set.

To generate a custom decoder, a generic decoder component is dynamically modified in response to a set of instruction definitions. Fig. 12, for example, illustrates a decoder icon 150 encapsulated within the instruction register icon 126 in the exemplary processing system, and representing a generic decoder component capable of being dynamically modified in a manner disclosed herein. The generic decoder component includes three ports initially: an "SR" port that receives an instruction load signal such as a CLOCK signal or a LOAD_INST signal, a "RESET" port configured to receive a

RESET signal, and an instruction ("INSTR") port configured to receive instructions from the data bus.

Fig. 13 illustrates a decoder customization process 160 that a user may undertake to generate a custom decoder from the generic component. The process begins in block 162 by creating an instruction set file, representing the definitions for the instructions to be executed by the modeled processor. Typically, the instruction set file is in a textual format, including definition of the accessible registers for the processor, as well as the supported instructions coupled with the signals to propagate in response to execution of such instructions. A wide variety of alternate instruction set formats may be used, and the invention is not limited to the particular implementation discussed herein.

Next, in block 164, the generic decoder icon discussed above is dropped into the existing design. Then, in block 166, the instruction set file is linked to the decoder component instance. Linking the instruction set file may be performed, for example, by selection of the generic decoder icon to open a parameter configuration window, and entering the filename of the instruction set file in the appropriate selection box. In response to submission of the instruction set file name to the decoder component instance, control passes to block 168 to execute a generate decoder routine, e.g., upon dismissal of the parameter configuration window by the user. The generate decoder routine is therefore dynamically called in response to linking of the generic decoder to the instruction set file. Dynamic generation of an instruction decoder may be implemented in a number of alternate manners, e.g., in response to specific user input to generate the decoder once an instruction set file has been linked thereto.

As will be discussed below, the generate decoder routine analyzes the instruction set file and creates a custom decoder icon including appropriate ports representing the signals to be asserted selectively by the instruction decoder in response to the various instructions received thereby. Once the customized decoder is generated, control passes to block 170, where the decoder is coupled to the rest of the design, e.g., by the user selecting the newly-created ports and coupling the ports to suitable ports for existing components in the model. Next, as represented at block 172, a user may start simulation of the model, with any errors resulting in the user editing the instruction set file as shown at block 174 and regenerating the decoder by passing control back to routine 168. Halting of the simulation without error results in the completion of process 160.

Generate decoder routine 168 is illustrated in greater detail in Fig. 14. Routine 168 begins in block 180 by initializing a decoder table to allocate memory for the table. Next, block 182 opens the specified instruction set file, and block 184 scans through the instruction set file to locate a register definition statement that defines the available registers that may be accessed by the decoder. In response to the located register definition, the available registers are defined. Moreover, if no register definition statement is detected, an error may be signaled. Also, if at a later point an instruction attempts to access a non-defined register, an error may be signaled to the user.

Next, block 186 locates the first instruction definition within the instruction set file. Block 188 determines whether the attempt to find such a definition was successful. If so, control passes to block 190 to retrieve each line of the instruction definition. In the illustrated embodiment, each line includes a signal (control line) and associated value to propagate on the signal (output type) in response to execution of the instruction.

Block 192 determines whether any lines were found for the first instruction definition. If not, an error is signaled and routine 168 terminates. If, however, at least one line is detected, control passes to block 194 to obtain the control line and output type from the first and second fields of a line in the instruction set definition.
Block 196 next determines whether the control line is already defined in the decoder table. If no such line is found in the table, control passes to block 198 to add the control line name and output type to the table. Control then passes to block 200 to determine if any more lines are present in the current instruction definition. If so, control returns to block 194 to process the next line in the instruction definition.

Once all lines in the current instruction definition have been processed, block 200 passes control to block 202 to locate the next instruction definition. Control then passes to block 188 to determine whether such definition was found. If so, control proceeds to block 190 to process the instruction in the manner discussed above. If, however, no additional definitions are found, control passes to block 204 to determine whether the table is empty (i.e., no instruction definitions were found in the file). If so, an error is signaled, and the routine is terminated. If not, however, control passes to block 206 to create a new decoder icon including port controls and labels for each of the control lines listed in the decoder table. Routine 168 is then complete.

To further illustrate the decoder customization process, Table IV below shows and exemplary instruction set file that may be used to generate a custom decoder from the generic decoder component described above:

**Table IV: Example Instruction Set Definition**

```
<rn    0|1|2|3|4|5|6|7>          ;defines 8 available registers

add ri,rj,rn                 ;defines "add" instruction
    alu_codop add                ;control signal to ALU is "add"
    source_1  ri                 ;first source is register ri
    source_2  rj                 ;second source is register rj
    dest      rn                 ;destination is register rn

sub ri,rj,rn                 ;defines "subtract" instruction
    alu_codop sub            ;control signal to ALU is "sub"
    source_1  ri                 ;first source is register ri
    source_2  rj                 ;second source is register rj
    dest      rn                 ;destination is register rn
```

The resulting decoder icon 150′ is illustrated in Fig. 15, whereby a port for an ALU control signal ALU_CODEOP is illustrated, along with three register control signals (SOURCE_1, SOURCE_2, and DEST) for coupling to a register file have been added to the generic decoder icon.

During simulation, when a signal rising edge occurs on the SR input port of the decoder, the decoder is configured to capture the value presented on the INSTR input port. Decoding of the captured value is then performed.

As an example, assume a value of "add r2, r3, r7" is captured at the INSTR input port of the generated decoder implementing the instruction set defined in Table IV above. The decoder is configured to first check if the first field of the received instruction "add" exists in the instruction set definition file. If not, the simulator is stopped with an error flag. Otherwise, an "add" value is propagated to the ALU_CODOP output port of the decoder.

The decoder is configured to thereafter check if the second field "r2" exists in the instruction set definition file. If not, the simulator is stopped with an error flag. If so, however, the value "r2" is propagated to the SOURCE_1 output port of the decoder. Likewise, the values "r3" and "r7" are propagated to the SOURCE_2 and DEST output ports of the decoder. It is to be noted that if a register "r9" that is not declared in the register definition is present in an instruction, an error will be signaled and the simulation halted.

In the illustrated implementation, the decoder component is configured to be sensitive to a rising edge event on the SR input port. Decoding is performed by string comparison, with any mismatches from the instruction definition format resulting in the generation of error flags and halts to the simulation, thereby enabling a developer to verify that developed code is syntactically correct. It will be appreciated that the development of suitable routines for comparing string instructions to an instruction set definition as described herein would be well within the ability of one of ordinary skill in the art having the benefit of the instant disclosure.

As discussed above, during simulation typically one or more observer or output components (e.g., component 136 of Fig. 10) are also used to output current state information for associated ports. These components are simply configured to observe events on associated ports and output current state information thereof to the display. In addition, in some embodiments check boxes or like controls may be selected on certain components to permit state information therefor to be captured and displayed in a debug window for logging to the display.

Fig. 16 illustrates yet another mechanism through which simulation data may be displayed to a user. In particular, the contents of memories such as data memories and program memories may be displayed in table or list form to a user, e.g., via windows 220 and 222. Each window 220, 222 includes a plurality of memory location entries 224, 226, with each entry identifying a memory location and the contents thereof. The current or last memory location accessed in the memory may be represented, for example, by a highlight bar 228, 230 or other appropriate distinctive display element, so that a user can readily determine, e.g., in the case of the program memory, what instruction is currently being executed during simulation. It will also be appreciated that since actual binary values are not actually required for simulation of a model as described herein, the assembly-language instructions may be displayed in textual format in each entry 226.

Important benefits of the aforementioned decoder implementation are reusability, extensibility and optimization of decoder designs, since an instruction decoder component can be readily modified simply through editing of the instruction set file and regeneration via routine 168. Such an environment therefore permits developers to quickly test out the benefits of adding support for new instructions in an existing design simply through the addition of new instruction definitions in an instruction file and modification of the test program used during simulation to incorporate such instructions.

Additional functionality may also be supported in the dynamic generation routine and/or the simulation routine to provide verification of an instruction decoder. For example, when a new instruction is added, material dependencies may be checked to verify if physically all services can be called concurrently or sequentially as commanded by the instruction. For example, some architectures may permit concurrent loading of multiple data points over multiple, independent memory ports, whereby other architectures may multiplex multiple data point requests over the same memory port. In the latter instance, concurrent requests for multiple data points over the same memory port may not be permissible, and an error may be signaled to a developer in response to dynamic generation to incorporate an instruction that attempts to do so.

In addition, it may be desirable to incorporate information about a pipeline structure for use in simulation, by adding delays that represent pipeline stages between a sequential execution of several services. Then, by combining instruction set definition information and such structural constraints, a cycle-accurate simulation may be obtained.

## Virtual Components

As mentioned above, in the illustrated implementation, native circuit components are typically defined in the Java programming language. Nonetheless, it may be desirable in some applications to further support the usage of additional components defined in non-native formats. For example, pre-existing circuit components may be defined in languages such as c, Esterel, VHDL, Verilog, etc. Rather than requiring such pre-existing circuit components to be recreated in a Java format, it may be desirable to support a "virtual" component architecture whereby a non-native component is encapsulated with suitable interfaces to enable integration of such a non-native component into rapid prototyping environment 41.

In particular, the Java programming language supports functions that allow a user to send variables to and capture results from an external program. To use such functions, which are often referred to as external functions, a developer is required to generate a Java program that sends variables to a specified external component, calls functions that are executed inside the external component, and capture the results produced by the external component. Where an external component is written in c or another language where variable types are important, strict adherence must be provided to variable types of an encapsulated external component.

Fig. 17 represents the basic structure of a virtual component 240 consistent with the invention. Encapsulated within virtual component 240 is an external component 242, typically written in a non-native (e.g., non-Java) language. Variables are provided to the external component from environment 41 via a port 244, where the variables are formatted and passed along by an input routine 246. Likewise, functions are initiated on the external component from environment 41 via a port 248, where such function initiations are formatted and passed along by a functions routine 250. Output data generated by the external component is captured by an output routine 252 and output from the virtual component via a port 254.

Various modifications may be made to the illustrated embodiments without departing from the spirit and scope of the invention. For example, other user interface methodologies and other programming environments may be used in the alternative.

Other modifications will be apparent to one of ordinary skill in the art. Therefore, the invention lies in the claims hereinafter appended.

What is claimed is:

1. A method of modeling an electronic circuit design on a computer, the method comprising:

(a) constructing a model of an electronic circuit design from first and second object-oriented circuit components, the first and second circuit components including first and second ports, respectively; and

(b) interfacing the first and second circuit components to one another through the first and second ports, wherein the first and second ports are configured to propagate both a first value for a first signal and a second value for a second signal between the first and second ports responsive to an event directed to at least one of the first and second ports.

2. The method of claim 1, further comprising performing simulation of the model, including:

(a) generating the event; and

(b) propagating the first and second values between the first and second ports in response to the event.

3. The method of claim 1, wherein the event includes an event name and an event trigger, the event trigger identifying a condition associated with the event name that triggers the event.

4. The method of claim 3, wherein the event name comprises a signal identifier, and wherein the condition for the event trigger is selected from the group consisting of a logic one, a logic zero, a rising edge, a falling edge and combinations thereof.

5. The method of claim 1, wherein constructing the model includes:

(a) receiving user input via a graphical user interface to add the first and second circuit components to the workspace; and

(b) displaying first and second icons on a computer display, the first and second icons respectively associated with the first and second circuit components, and the first and second icons respectively including first and second port controls associated with the first and second ports.

6. The method of claim 5, wherein interfacing the first and second circuit components to one another includes:

(a) receiving user input via the graphical user interface to interface the first and second circuit components to one another; and

(b) displaying an interconnect element on the display extending between the first and second port controls to collectively represent propagation of the first and second signals between the first and second ports.

7. The method of claim 6, wherein the interconnect element comprises a single graphical line extending between the first and second port controls.

8. The method of claim 1, wherein the first circuit component encapsulates third and fourth circuit components, the third and fourth circuit components including third and fourth ports, respectively, and wherein the first port is further configured to propagate the first value between the second port and the third port, and to propagate the second value between the second port and the fourth port, responsive to the event.

9. The method of claim 1, wherein the first value comprises a symbol.

10. The method of claim 9, wherein the first value comprises a string.

11. An apparatus, comprising:

(a) a memory configured to store a model of an electronic circuit design; and

(b) a program configured to construct the model from first and second object-oriented circuit components that include first and second ports, respectively, and to interface the first and second circuit components to one another through the first and second ports, wherein the first and second ports are configured to propagate both a first value for a first signal and a second value for a second signal between the first and second ports responsive to an event directed to at least one of the first and second ports.

12. The apparatus of claim 11, wherein the program is further comprised to perform simulation of the model, wherein simulation of the model includes generation of the event and propagation of the first and second values between the first and second ports in response to the event.

13. The apparatus of claim 11, wherein the event includes a signal identifier and an event trigger, the event trigger identifying a condition associated with the event name that triggers the event, the condition selected from the group consisting of a logic one, a logic zero, a rising edge, a falling edge and combinations thereof.

14. The apparatus of claim 11, wherein the program is configured to construct the model by receiving user input via a graphical user interface to add the first and second circuit components to the workspace, and displaying first and second icons on a computer display, the first and second icons respectively associated with the first and second circuit components, and the first and second icons respectively including first and second port controls associated with the first and second ports, and wherein the program is configured to interface the first and second circuit components to one another by receiving user input via the graphical user interface to interface the first and second circuit components to one

another, and displaying an interconnect element on the display extending between the first and second port controls to collectively represent propagation of the first and second signals between the first and second ports.

15. The apparatus of claim 11, wherein the first circuit component encapsulates third and fourth circuit components, the third and fourth circuit components including third and fourth ports, respectively, and wherein the first port is further configured to propagate the first value between the second port and the third port, and to propagate the second value between the second port and the fourth port, responsive to the event.

16. The apparatus of claim 11, wherein the first value comprises a string.

17. A program product, comprising:

(a) a program configured to construct a model of an electronic circuit design from first and second object-oriented circuit components that include first and second ports, respectively, and to interface the first and second circuit components to one another through the first and second ports, wherein the first and second ports are configured to propagate both a first value for a first signal and a second value for a second signal between the first and second ports responsive to an event directed to at least one of the first and second ports; and

(b) a signal bearing medium bearing the program.

18. The program product of claim 17, wherein the signal bearing medium includes at least one of a recordable medium and a transmission medium.

19. A method of modeling an electronic circuit design on a computer, the method comprising:

(a) receiving at least one instruction definition that defines an instruction capable of being executed by a processing system model; and

(b) processing the instruction definition to configure an instruction decoder circuit component for the processing system model to simulate execution of the instruction.

20. The method of claim 19, wherein processing the instruction definition includes dynamically generating the instruction decoder circuit component from a generic instruction decoder circuit component.

21. The method of claim 20, further comprising:

(a) modifying an icon representation of the generic instruction decoder circuit component responsive to the instruction definition; and

(b) displaying the modified icon representation on a computer display.

22. The method of claim 19, wherein the instruction definition includes a plurality of values to be propagated for a plurality of signals responsive to execution of the instruction.

23. The method of claim 22, further comprising performing simulation on the model by propagating the plurality of values in response to receipt of an instruction meeting the instruction definition by the instruction decoder circuit component during simulation.

24. The method of claim 23, wherein the instruction definition is identified by a text string, and wherein performing simulation on the model includes interpreting the text string.

25. The method of claim 22, wherein the instruction decoder circuit component includes a first port associated with a first signal from the plurality of signals, the method further comprising interfacing the instruction decoder circuit component with a second circuit component that includes a second port by coupling the first port to the second port.

26. The method of claim 19, wherein the instruction definition is defined within an instruction set definition file including a plurality of instruction definitions, wherein processing the instruction definition includes processing the instruction set definition file to configure the instruction decoder circuit component to simulate execution of the plurality of instructions.

27. The method of claim 26, wherein the instruction set definition file further includes a register definition, the method further comprising processing the register definition to identify available registers in the processing system.

28. The method of claim 26, further comprising modifying the instruction decoder circuit component in response to a modification to the instruction set definition file.

29. The method of claim 28, wherein modifying the instruction decoder circuit component comprises reprocessing the instruction set definition file after modification to the instruction set definition file.

30. The method of claim 26, wherein each instruction in the instruction set definition file identifies at least one value to be propagated for at least one of a plurality of signals responsive to execution of such instruction, and wherein processing the instruction set definition file includes building a decoder table comprising a plurality of entries, each entry identifying a signal from the plurality of signals.

31. The method of claim 26, wherein each instruction definition in the instruction set definition file includes a text identifier identifying the instruction in an assembly language format.

32. An apparatus, comprising:

(a) a memory configured to store a processing system model; and

(b) a program configured to receive at least one instruction definition that defines an instruction capable of being executed by the processing system model, and to process the instruction definition to configure an instruction decoder circuit component for the processing system model to simulate execution of the instruction.

33. The apparatus of claim 32, wherein the program is configured to process the instruction definition by dynamically generating the instruction decoder circuit component from a generic instruction decoder circuit component.

34. The apparatus of claim 33, wherein the program is further configured to modify an icon representation of the generic instruction decoder circuit component responsive to the instruction definition, and to display the modified icon representation on a computer display.

35. The apparatus of claim 32, wherein the instruction definition includes a plurality of values to be propagated for a plurality of signals responsive to execution of the instruction.

36. The apparatus of claim 35, wherein the program is further configured to perform simulation on the model by propagating the plurality of values in response to receipt of an instruction meeting the instruction definition by the instruction decoder circuit component during simulation.

37. The apparatus of claim 36, wherein the instruction definition is identified by a text string, and wherein the program is configured to perform simulation on the model by interpreting the text string.

38. The apparatus of claim 35, wherein the instruction decoder circuit component includes a first port associated with a first signal from the plurality of signals, the program further configured to interface the instruction decoder circuit component with a second circuit component that includes a second port by coupling the first port to the second port.

39. The apparatus of claim 32, wherein the instruction definition is defined within an instruction set definition file including a plurality of instruction definitions, wherein the program is configured to process the instruction definition by processing the instruction set definition file to configure the instruction decoder circuit component to simulate execution of the plurality of instructions.

40. The apparatus of claim 39, wherein the program is configured to modify the instruction decoder circuit component in response to a modification to the instruction set definition file.

41. The apparatus of claim 40, wherein the program is configured to modify the instruction decoder circuit component by reprocessing the instruction set definition file after modification to the instruction set definition file.

42. The apparatus of claim 39, wherein each instruction in the instruction set definition file identifies at least one value to be propagated for at least one of a plurality of signals responsive to execution of such instruction, and wherein the program is configured to process the instruction set definition file by building a decoder table comprising a plurality of entries, each entry identifying a signal from the plurality of signals.

43. A program product, comprising:

(a) a program configured to receive at least one instruction definition that defines an instruction capable of being executed by a processing system model, and to process the instruction definition to configure an instruction decoder circuit component for the processing system model to simulate execution of the instruction; and

(b) a signal bearing medium bearing the program.

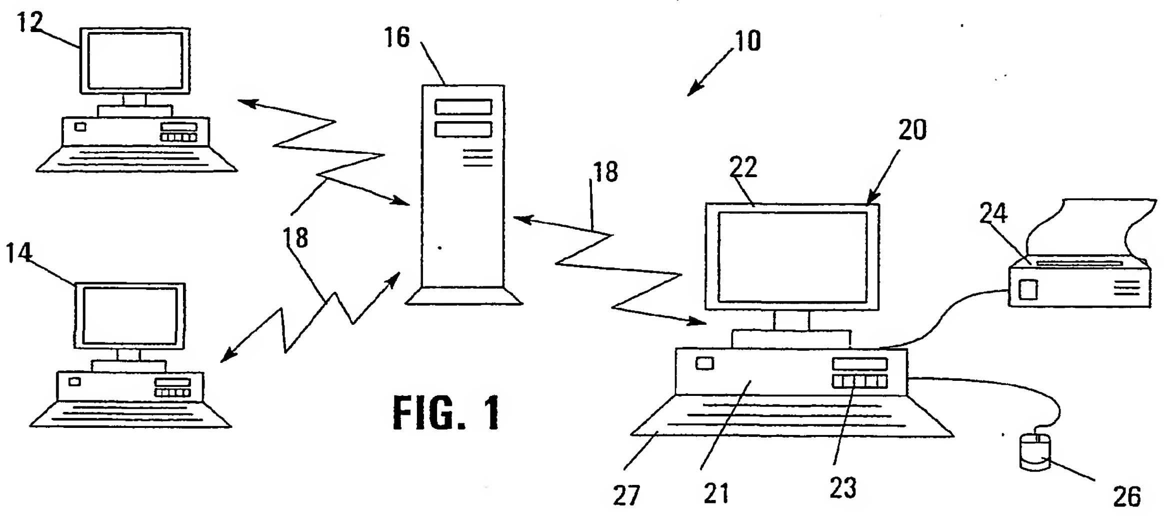


FIG. 1

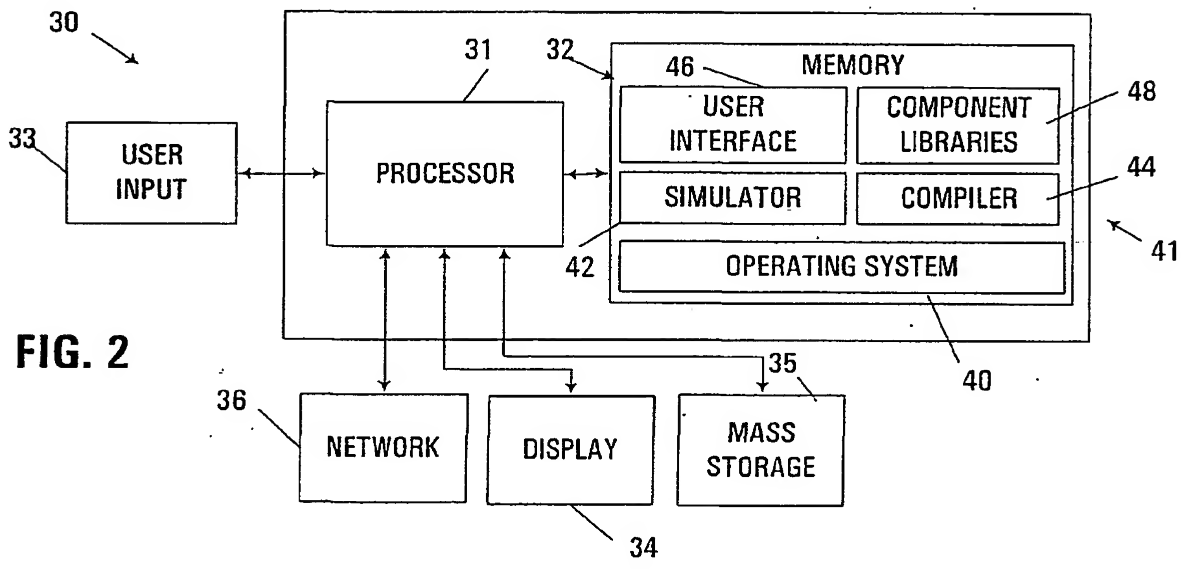


FIG. 2

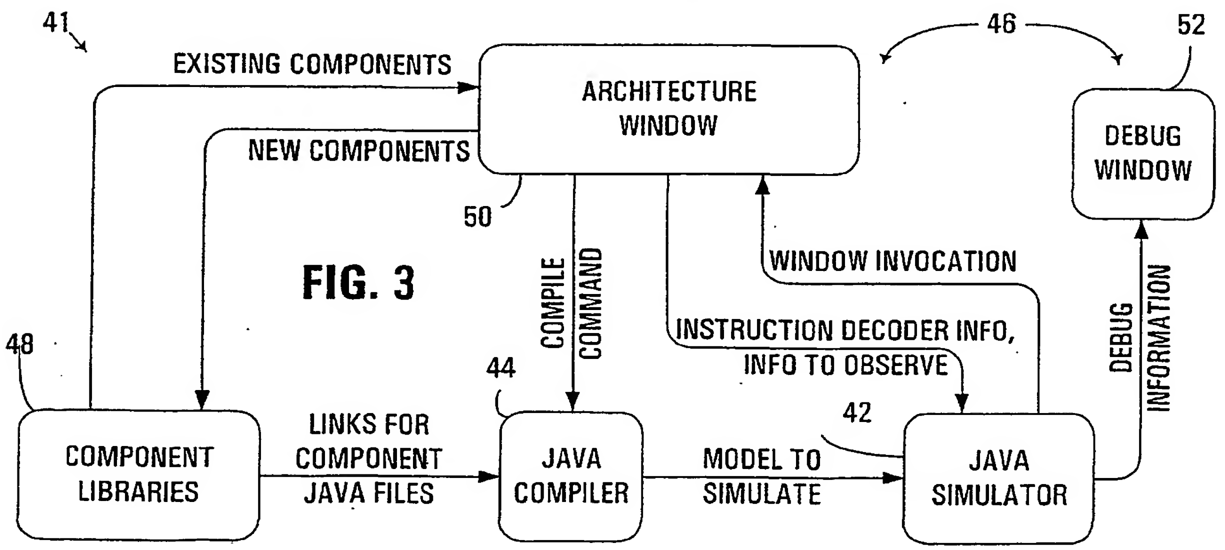


FIG. 3

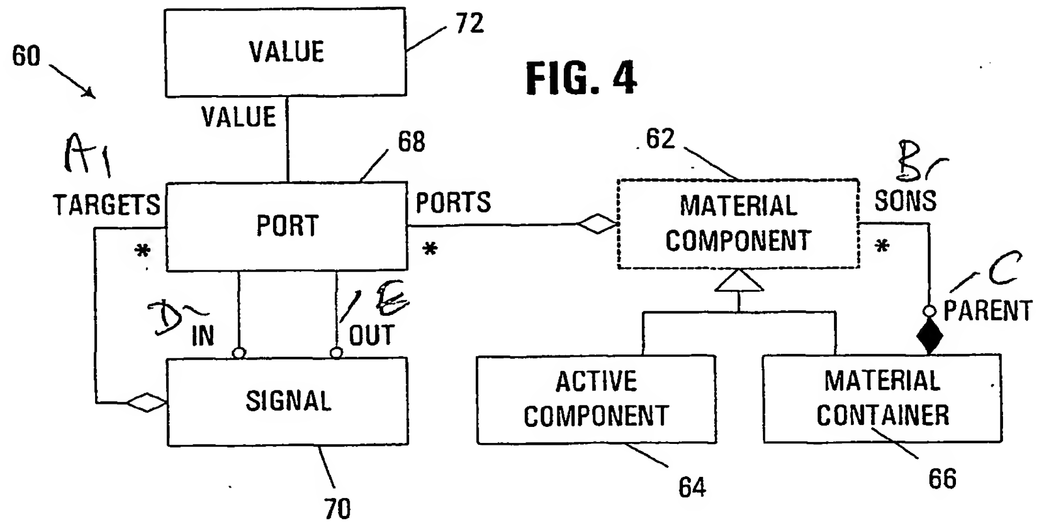


FIG. 4

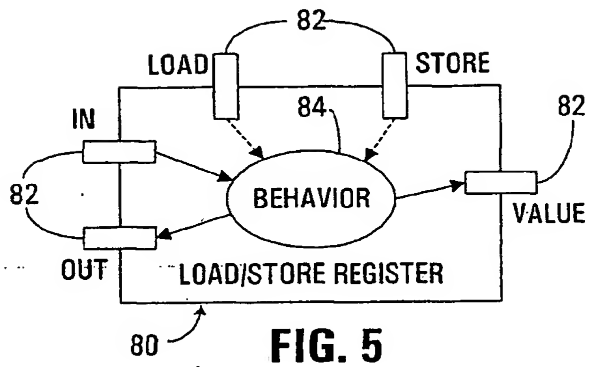


FIG. 5

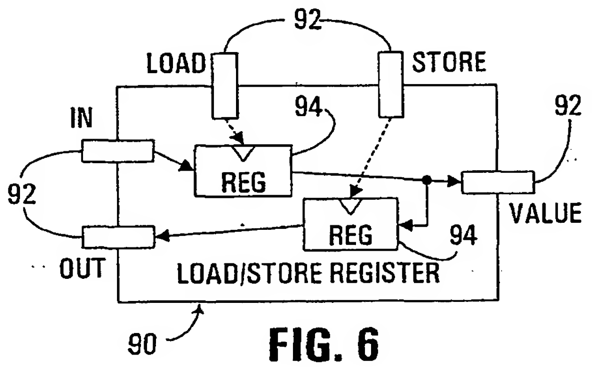


FIG. 6

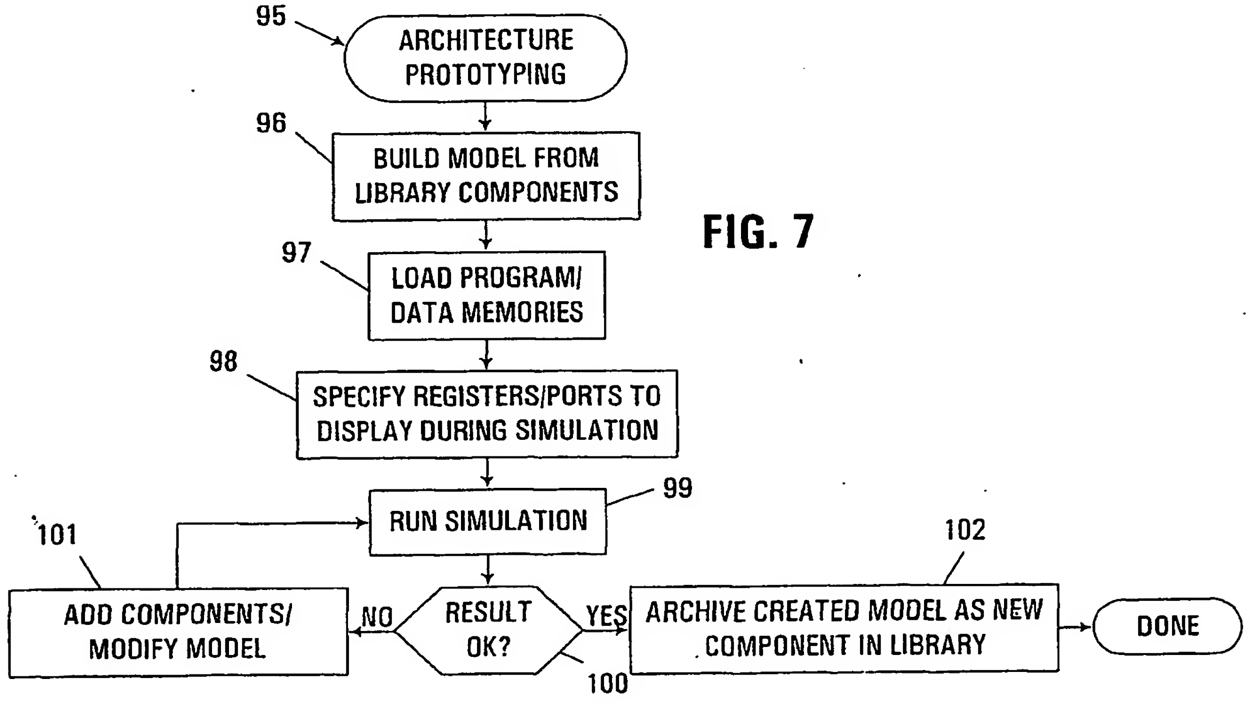

95
ARCHITECTURE PROTOTYPING
96
BUILD MODEL FROM LIBRARY COMPONENTS
97
LOAD PROGRAM/ DATA MEMORIES
98
SPECIFY REGISTERS/PORTS TO DISPLAY DURING SIMULATION
99
RUN SIMULATION
101
ADD COMPONENTS/ MODIFY MODEL
NO
RESULT OK?
YES
100
102
ARCHIVE CREATED MODEL AS NEW COMPONENT IN LIBRARY
DONE

FIG. 7

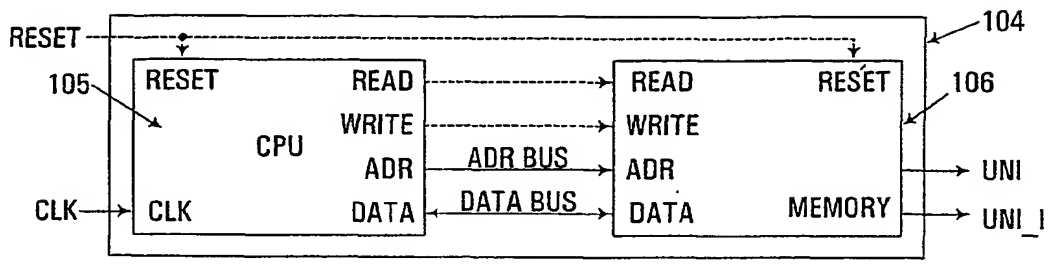

RESET
104
105
RESET
CPU
READ
WRITE
ADR
DATA
CLK
CLK
ADR BUS
DATA BUS
READ
WRITE
ADR
DATA
RESET
MEMORY
106
UNI
UNI_I

FIG. 8

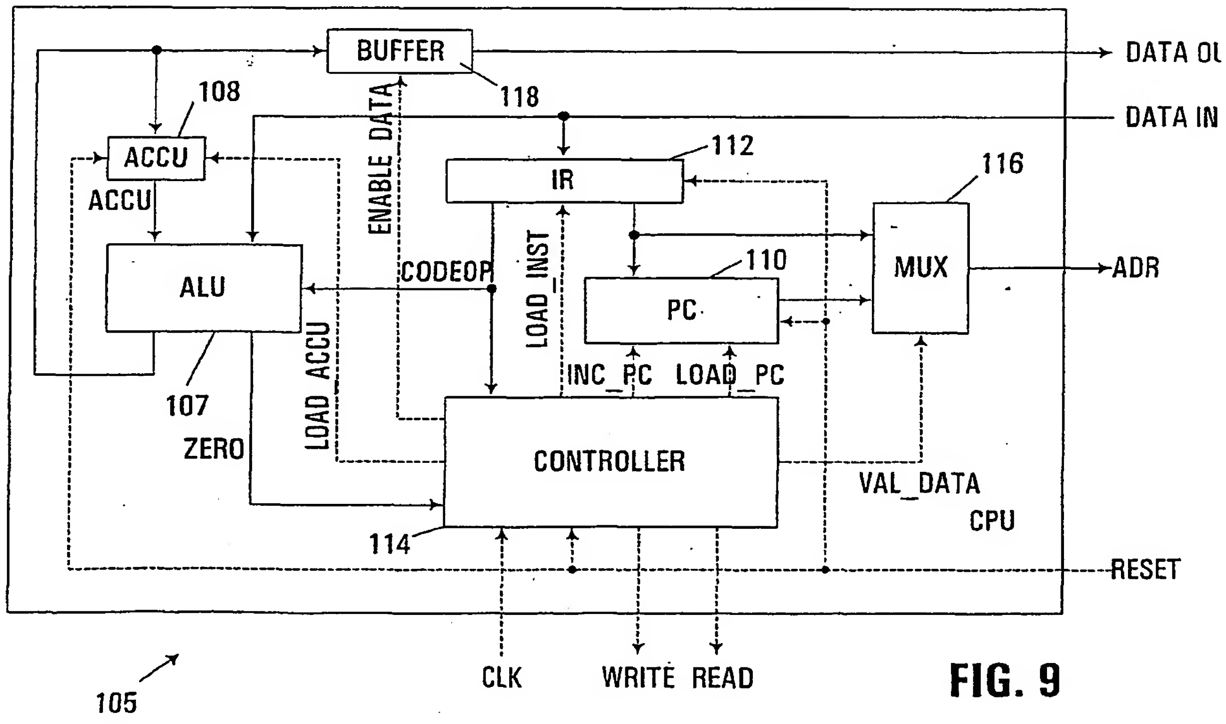

BUFFER
118
DATA OU
DATA IN
108
ACCU
ACCU
ENABLE DATA
IR
112
116
MUX
ADR
ALU
CODEOP
LOAD_INST
PC
110
LOAD ACCU
INC_PC
LOAD_PC
107
ZERO
CONTROLLER
VAL_DATA
CPU
114
RESET
CLK
WRITE
READ
105

FIG. 9

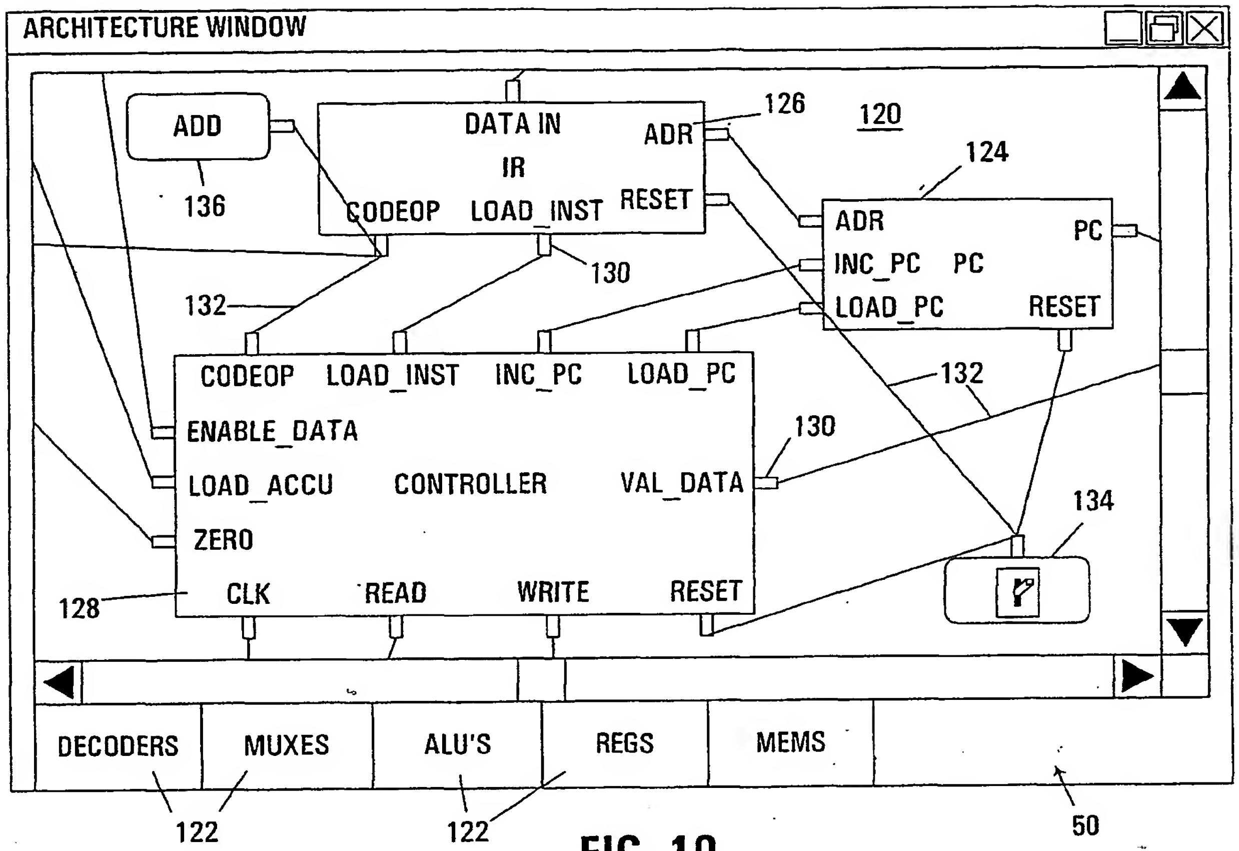

ARCHITECTURE WINDOW
ADD
136
DATA IN
IR
ADR
RESET
CODEOP
LOAD_INST
126
120
124
ADR
INC_PC
PC
LOAD_PC
PC
RESET
130
132
CODEOP
LOAD_INST
INC_PC
LOAD_PC
ENABLE_DATA
LOAD_ACCU
CONTROLLER
VAL_DATA
ZERO
CLK
READ
WRITE
RESET
128
130
132
134
DECODERS
MUXES
ALU'S
REGS
MEMS
122
122
50

FIG. 10

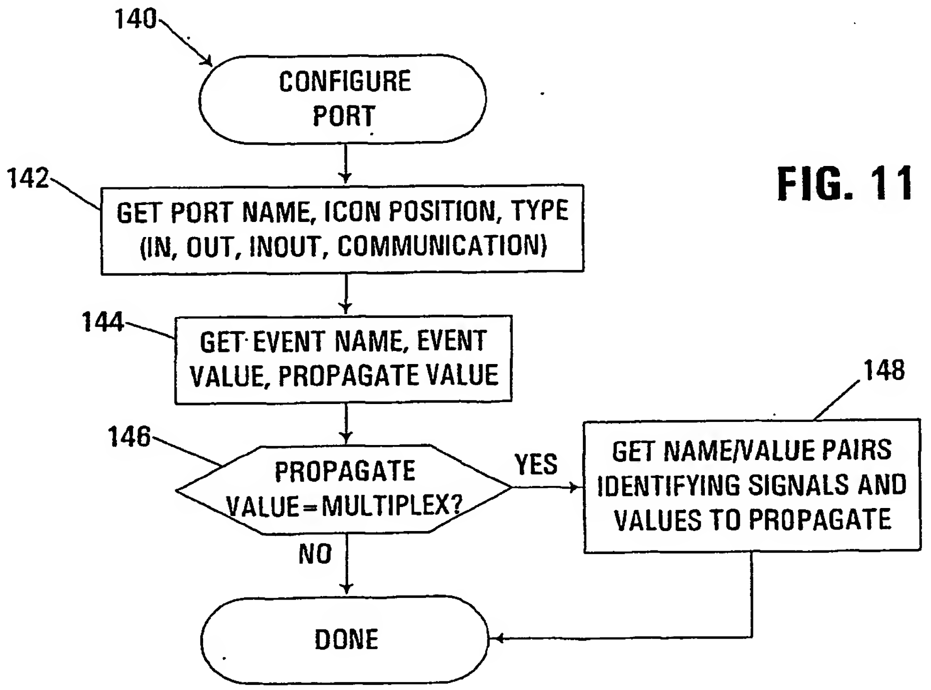

140
CONFIGURE PORT
142
GET PORT NAME, ICON POSITION, TYPE (IN, OUT, INOUT, COMMUNICATION)
144
GET EVENT NAME, EVENT VALUE, PROPAGATE VALUE
146
PROPAGATE VALUE=MULTIPLEX?
YES
NO
148
GET NAME/VALUE PAIRS IDENTIFYING SIGNALS AND VALUES TO PROPAGATE
DONE

FIG. 11

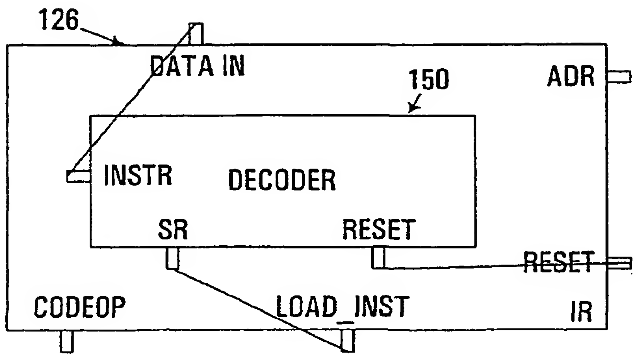


FIG. 12

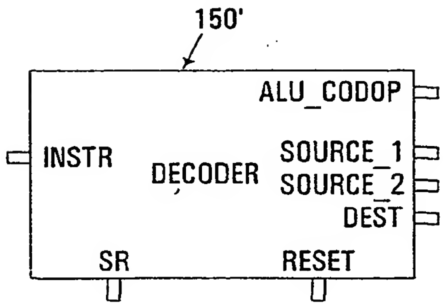


FIG. 15

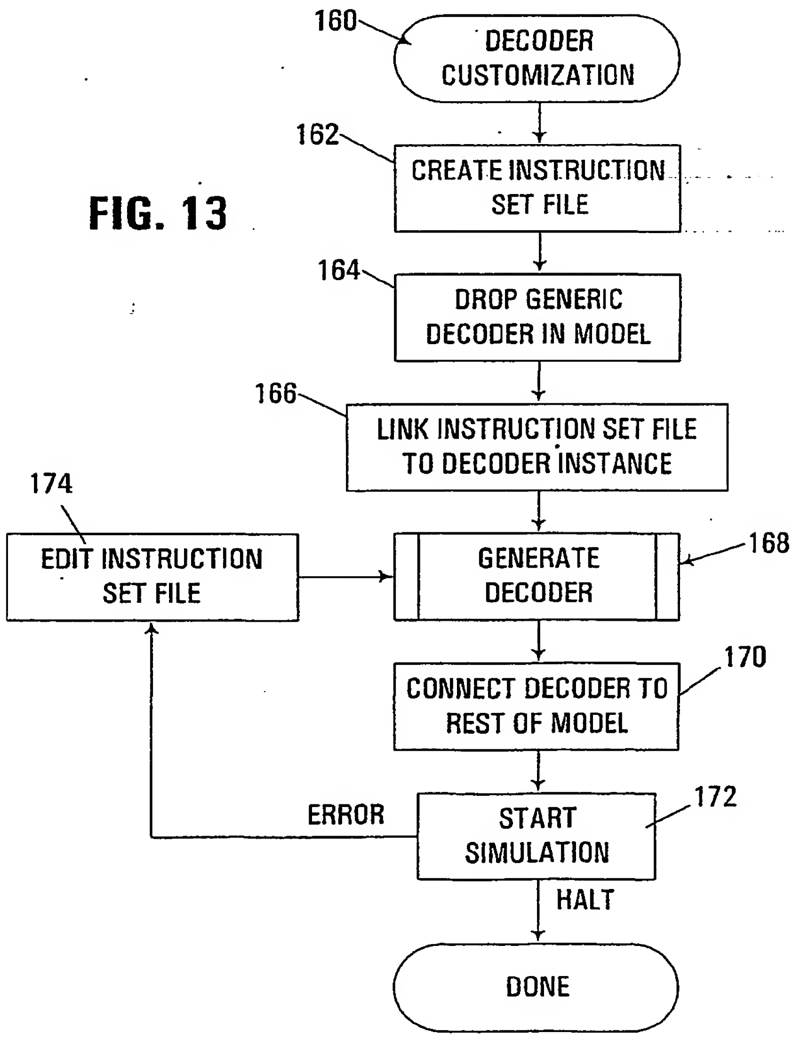

160
DECODER CUSTOMIZATION
162
CREATE INSTRUCTION SET FILE
164
DROP GENERIC DECODER IN MODEL
166
LINK INSTRUCTION SET FILE TO DECODER INSTANCE
174
EDIT INSTRUCTION SET FILE
GENERATE DECODER
168
CONNECT DECODER TO REST OF MODEL
170
ERROR
START SIMULATION
172
HALT
DONE

FIG. 13

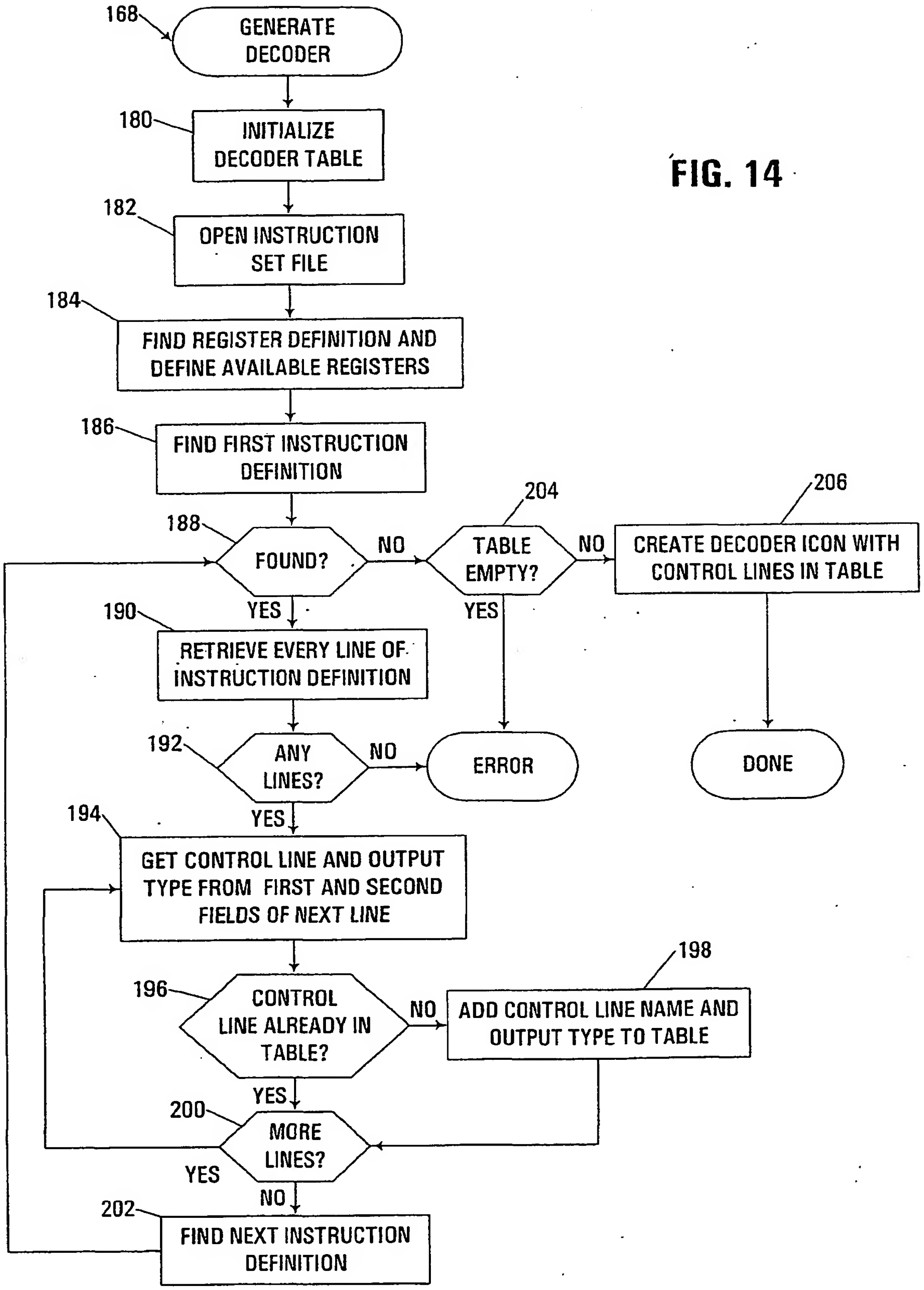
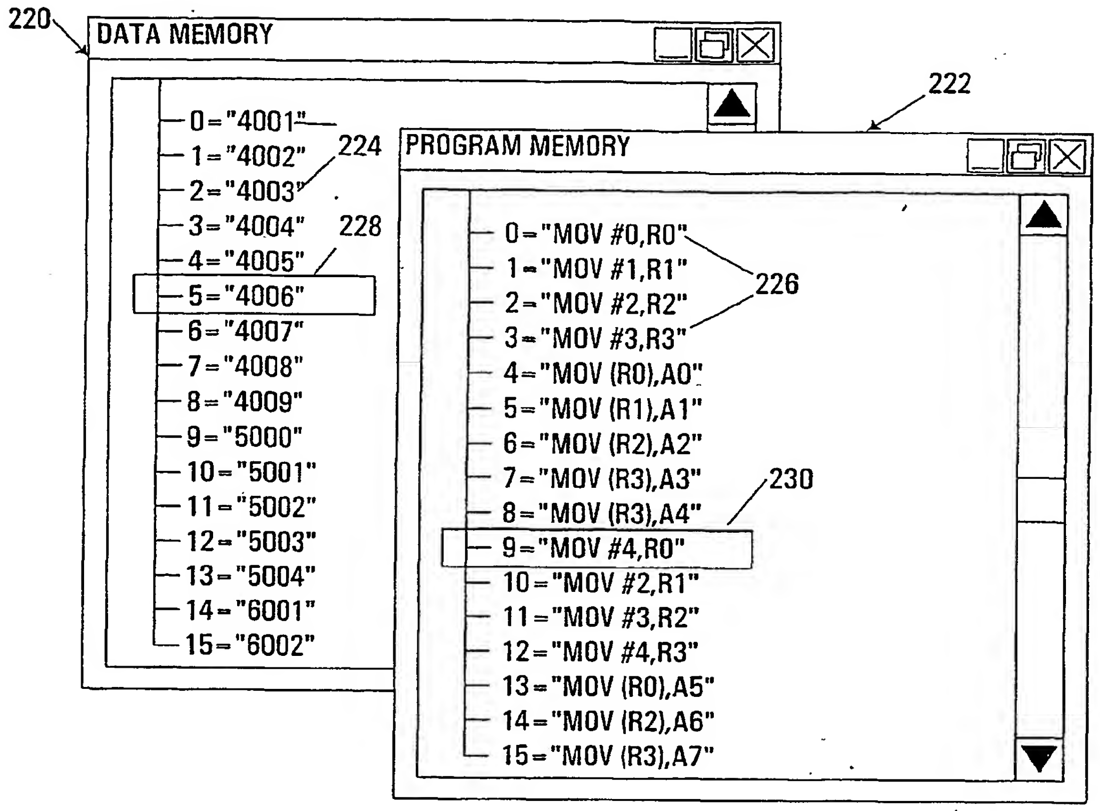

168
GENERATE DECODER
180
INITIALIZE DECODER TABLE
182
OPEN INSTRUCTION SET FILE
184
FIND REGISTER DEFINITION AND DEFINE AVAILABLE REGISTERS
186
FIND FIRST INSTRUCTION DEFINITION
188
FOUND?
NO
204
TABLE EMPTY?
NO
206
CREATE DECODER ICON WITH CONTROL LINES IN TABLE
YES
YES
190
RETRIEVE EVERY LINE OF INSTRUCTION DEFINITION
192
ANY LINES?
NO
ERROR
DONE
YES
194
GET CONTROL LINE AND OUTPUT TYPE FROM FIRST AND SECOND FIELDS OF NEXT LINE
196
CONTROL LINE ALREADY IN TABLE?
NO
198
ADD CONTROL LINE NAME AND OUTPUT TYPE TO TABLE
YES
200
MORE LINES?
YES
NO
202
FIND NEXT INSTRUCTION DEFINITION

FIG. 14



DATA MEMORY

```
0="4001"
1="4002"
2="4003"
3="4004"
4="4005"
5="4006"
6="4007"
7="4008"
8="4009"
9="5000"
10="5001"
11="5002"
12="5003"
13="5004"
14="6001"
15="6002"
```

PROGRAM MEMORY

```
0="MOV #0,R0"
1="MOV #1,R1"
2="MOV #2,R2"
3="MOV #3,R3"
4="MOV (R0),A0"
5="MOV (R1),A1"
6="MOV (R2),A2"
7="MOV (R3),A3"
8="MOV (R3),A4"
9="MOV #4,R0"
10="MOV #2,R1"
11="MOV #3,R2"
12="MOV #4,R3"
13="MOV (R0),A5"
14="MOV (R2),A6"
15="MOV (R3),A7"
```

FIG. 16

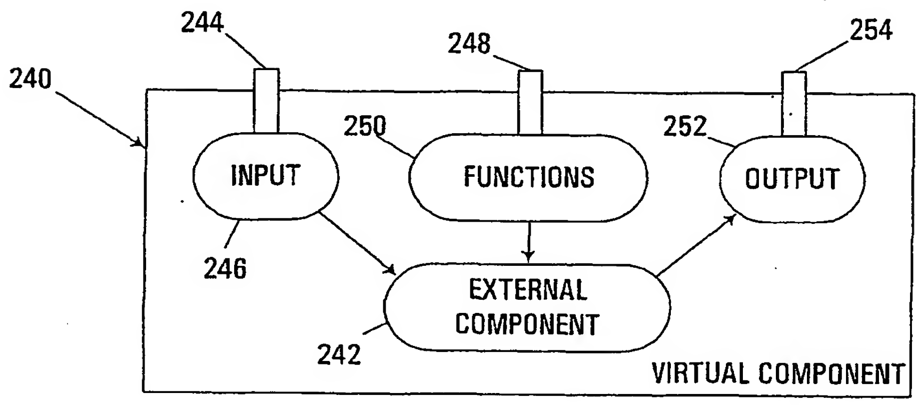


FIG. 17

(12) INTERNATIONAL APPLICATION PUBLISHED UNDER THE PATENT COOPERATION TREATY (PCT)

(19) World Intellectual Property Organization
International Bureau

(43) International Publication Date
29 November 2001 (29.11.2001)

PCT

(10) International Publication Number
WO 01/090961 A3

(51) International Patent Classification⁷: G06F 17/50

(21) International Application Number: PCT/US01/09106

(22) International Filing Date: 22 March 2001 (22.03.2001)

(25) Filing Language: English

(26) Publication Language: English

(30) Priority Data:
09/577,509 24 May 2000 (24.05.2000) US

(71) Applicant: KONINKLIJKE PHILIPS ELECTRONICS NV [NL/NL]; Groenewoudseweg 1, NL-5621 BA Eindhoven (NL).

(71) Applicant *(for all designated States except CN, JP, KR)*: PHILIPS ELECTRONICS NORTH AMERICA CORPORATION [US/US]; 1000 West Maude Avenue, Sunnyvale, CA 94086-2810 (US).

(72) Inventors: DUBOC, Jean, Francois; 2893, Moyenne Corniche des Pugets, F-06700 Saint-Laurent-du-Var (FR). MALLET, Frederic; 30, Boulevard Las Palmas, F-06100 Nice (FR). BOERI, Fernand; 131, Promenade des Anglais, F-06200 Nice (FR).

(74) Agents: STINEBRUNER, Scott, A. et al.; Wood, Herron & Evans, LLP, 2700 Carew Tower, Cincinnati, OH 45202 (US).

(81) Designated States *(national)*: CN, JP, KR.

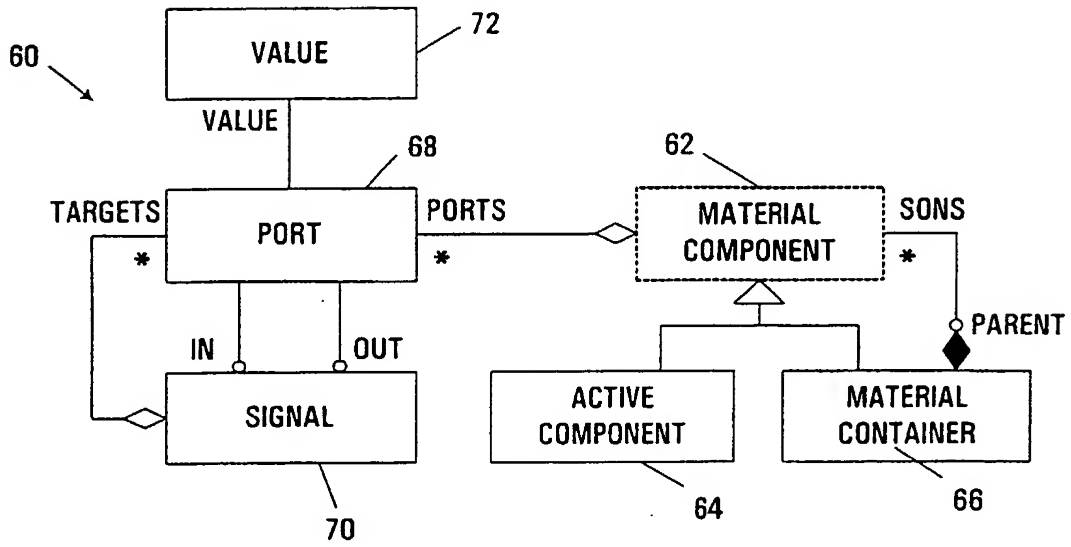
(84) Designated States *(regional)*: European patent (AT, BE, CH, CY, DE, DK, ES, FI, FR, GB, GR, IE, IT, LU, MC, NL, PT, SE, TR).

**Published:**
— *with international search report*
— *before the expiration of the time limit for amending the claims and to be republished in the event of receipt of amendments*

(88) Date of publication of the international search report: 5 December 2002

*[Continued on next page]*

(54) Title: ENHANCEMENTS TO OBJECT-ORIENTED ELECTRONIC CIRCUIT DESIGN MODELING AND SIMULATION ENVIRONMENT



(57) Abstract: An apparatus, program product and method enhance the modeling and stimulation of electronic circuit designs on computers, particularly in the area of object-oriented modeling and simulation. Multiplexed communication ports (130) may be supported that support the propagation of values for multiple signals in response to single events directed to such ports (130), thus logically combining multiple signal paths together into a single logical path. Dynamic instruction decoder generation (160) may also be supported, such that an instruction definition that defines an instruction capable of being executed by a processing system model may be utilized in the configuration of an instruction decoder circuit component (150') for the processing system model to stimulate execution of the instruction.

WO 01/090961 A3

# INTERNATIONAL SEARCH REPORT

ıational Application No
ı.ı/US 01/09106

**A. CLASSIFICATION OF SUBJECT MATTER**
IPC 7 G06F17/50

According to International Patent Classification (IPC) or to both national classification and IPC

**B. FIELDS SEARCHED**

Minimum documentation searched (classification system followed by classification symbols)
IPC 7 G06F

Documentation searched other than minimum documentation to the extent that such documents are included in the fields searched

Electronic data base consulted during the international search (name of data base and, where practical, search terms used)

EPO-Internal, INSPEC, IBM-TDB

**C. DOCUMENTS CONSIDERED TO BE RELEVANT**

| Category * | Citation of document, with indication, where appropriate, of the relevant passages | Relevant to claim No. |
|---|---|---|
| X | MALLET F ET AL: "Hardware architecture modelling using an object-oriented method" EUROMICRO CONFERENCE, 1998. PROCEEDINGS. 24TH VASTERAS, SWEDEN 25-27 AUG. 1998, LOS ALAMITOS, CA, USA,IEEE COMPUT. SOC, US, 25 August 1998 (1998-08-25), pages 147-153, XP010298090 ISBN: 0-8186-8646-4 the whole document<br>---<br>-/-- | 1-43 |

[✓ X] Further documents are listed in the continuation of box C. [ ] Patent family members are listed in annex.

° Special categories of cited documents :

"A" document defining the general state of the art which is not considered to be of particular relevance
"E" earlier document but published on or after the international filing date
"L" document which may throw doubts on priority claim(s) or which is cited to establish the publication date of another citation or other special reason (as specified)
"O" document referring to an oral disclosure, use, exhibition or other means
"P" document published prior to the international filing date but later than the priority date claimed

"T" later document published after the international filing date or priority date and not in conflict with the application but cited to understand the principle or theory underlying the invention
"X" document of particular relevance; the claimed invention cannot be considered novel or cannot be considered to involve an inventive step when the document is taken alone
"Y" document of particular relevance; the claimed invention cannot be considered to involve an inventive step when the document is combined with one or more other such documents, such combination being obvious to a person skilled in the art.
"&" document member of the same patent family

| | |
|---|---|
| Date of the actual completion of the international search<br>12 September 2002 | Date of mailing of the international search report<br>02/10/2002 |
| Name and mailing address of the ISA<br>European Patent Office, P.B. 5818 Patentlaan 2<br>NL – 2280 HV Rijswijk<br>Tel. (+31-70) 340-2040, Tx. 31 651 epo nl,<br>Fax: (+31-70) 340-3016 | Authorized officer<br>Guingale, A |

# INTERNATIONAL SEARCH REPORT

International Application No
…/US 01/09106

C.(Continuation) DOCUMENTS CONSIDERED TO BE RELEVANT

| Category ° | Citation of document, with indication,where appropriate, of the relevant passages | Relevant to claim No. |
|---|---|---|
| X | MALLET F ET AL: "Esterel and Java in an object-oriented modelling and simulation framework for heterogeneous software and hardware systems. The SEP approach" PROCEEDINGS 25TH EUROMICRO CONFERENCE. INFORMATICS: THEORY AND PRACTICE FOR THE NEW MILLENNIUM, PROCEEDINGS OF EUROMICRO WORKSHOP, MILAN, ITALY, 8-10 SEPT. 1999, pages 214-221 vol.1, XP002213241 1999, Los Alamitos, CA, USA, IEEE Comput. Soc, USA ISBN: 0-7695-0321-7 | 1-18 |
| A | paragraphs '02.1!,'02.2!,'02.3! figures 1,2 | 19-43 |

-----